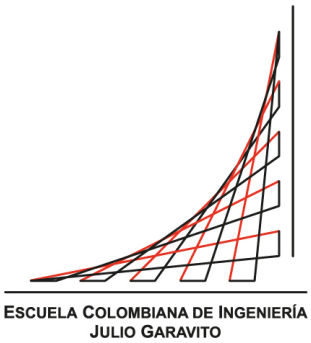
Sistema de visión por computador embebido para la detección y el seguimiento de objetivo móvil desde un vehículo aéreo no tripulado (UAV) usando ROS



David Bohorquez Caro

Escuela Colombiana de Ingeniería Julio Garavito
Maestría en Ingeniería Electrónica
Bogotá, Colombia
2021

Sistema de visión por computador embebido para la detección y el seguimiento de objetivo móvil desde un vehículo aéreo no tripulado (UAV) usando ROS

David Bohorquez Caro

Trabajo de grado presentado como requisito para optar al título de:
Magíster en Ingeniería Electrónica, con énfasis en Control y Automatización Industrial

Director:
Ing. Alexander Pérez Ruíz, M.Sc., Ph.D.

Escuela Colombiana de Ingeniería Julio Garavito
Maestría en Ingeniería Electrónica
Bogotá, Colombia
2021

A mi Padre en los cielos
y a mi madre en la tierra.

Agradecimientos

Quiero empezar por agradecer a mi esposa Tatiana, por su apoyo incondicional, comprensión y motivación durante estos años de estudio. A mi madre Carmen, por su amor y cariño en todo tiempo, por escucharme y aconsejarme en los momentos difíciles. A mis hermanos Alexander y Catherinne, por sus palabras que me incentivan a crecer en cada aspecto de mi vida.

A mi director de trabajo de grado el ingeniero Alexander Pérez, por su guía y paciencia durante todo el proceso de aprendizaje en el programa. Su capacidad para compartir conocimientos y enseñanzas, así como los aportes y consejos recibidos, han sido fundamentales para la culminación de este trabajo de grado. Aprecio las numerosas reuniones para discutir acerca de los avances del proyecto y las revisiones al documento.

Por último quiero agradecer, a la Escuela Colombiana de Ingeniería Julio Garavito por permitirme cursar el programa de Maestría en Ingeniería Electrónica con énfasis en Control y Automatización Industrial, porque gracias a la calidad educativa y el prestigio de la institución, se han abierto nuevas oportunidades en el ámbito laboral y profesional.

Resumen

El objetivo principal de este proyecto de grado es desarrollar un sistema de visión por computador embebido para dotar a un cuadricóptero con la capacidad de detectar y hacer el seguimiento autónomo de un objeto móvil terrestre visto desde la plataforma aérea no tripulada (*Unmanned Aerial Vehicle (UAV)*), utilizando ROS. La detección y el seguimiento de objetos en movimiento usando plataformas UAV, son consideradas tareas difíciles para los sistemas de visión por computador y su implementación requiere superar varios desafíos.

Factores como la variación del punto de vista, la deformación del objeto, oclusión completa o parcial, condiciones de iluminación, fondo desordenado o visualmente complejo y la velocidad del objeto. Los algoritmos y detectores de objetos propuestos en este trabajo de grado buscan resolver estos problemas asociados con la detección y el seguimiento visual de un carro de radio control en movimiento visto desde un UAV en un ambiente exterior y libre de obstáculos. Dado que el principal requerimiento del sistema de visión embebido, es la necesidad de obtener resultados en tiempo real, se exploran algunos detectores de objetos *deep learning* y se evalúan métricas comunes como el rendimiento y la precisión, y se calcula la velocidad máxima a la que puede desplazarse el carro de radio control, al final se comparan los resultados obtenidos con cada modelo de aprendizaje profundo entrenado. Para el entrenamiento en la detección del objeto seleccionado, se conformó un *dataset* compuesto por un conjunto de imágenes y videos del carro de radio control en ambientes interiores y exteriores reales, con condiciones ambientales favorables. El orden y las texturas del fondo del escenario fueron seleccionados de forma aleatoria y diversa.

En la revisión del estado del arte, la mayoría de enfoques propuestos implican realizar el procesamiento de imágenes en una estación ubicada en tierra. Sin embargo, otro requerimiento, consiste en conformar un sistema de visión de tal modo que pueda ser transportado por un UAV. Implementar aplicaciones *deep learning* requiere de un alto costo computacional, por lo que ejecutar modelos de aprendizaje profundo sobre un computador embebido, representa un desafío por los limitantes en recursos de procesamiento y *hardware*. Para lograr la portabilidad del sistema, se realizó un análisis de diferentes componentes *hardware* y *software*, y fueron seleccionados aquellos que permiten cumplir con el requisito de realizar el procesamiento de

imágenes y las tareas de visión desde un sistema embebido que puede ir a bordo de un UAV.

El *middleware* ROS proporciona funcionalidad para abstracción de *hardware*, controladores de dispositivos, comunicación de procesos entre múltiples máquinas y que contiene diferentes herramientas para realizar pruebas y simulaciones. Este proyecto utilizó ROS como entorno de trabajo estandarizado para el desarrollo de la aplicación robótica, se diseñó e implementó un entorno de simulación *Hardware-in-the-loop (HIL)* de forma que el sistema embebido no controla un UAV real sino uno simulado. Para validar el funcionamiento del sistema, se implementó una interfaz en ROS que permite la comunicación entre el sistema de visión por computador embebido y un cuadricóptero. Se empleó el paquete de ROS *hector quadrotor*, el cual ofrece un modelo 3D de un cuadricóptero, con dinámicas de vuelo realistas, cámaras y sensores integrados. Una vez conformado el ambiente de simulación se realizaron diferentes vuelos, experimentos y pruebas con la ayuda de herramientas de visualización como Rviz (*ROS Visualization*) y Gazebo que se integran con ROS.

Este trabajo de grado concluye que los sensores ópticos proporcionan una fuente de información confiable y su procesamiento constituye una herramienta de navegación para las tareas comunes de los UAV. Los detectores de objetos entrenados con *Single Shot Multibox Detector* (SSD) presentan el mejor equilibrio entre las métricas evaluadas, con un rendimiento de $18.1 \sim 18.4$ FPS y una precisión entre $84.97\,\% \sim 94.44\,\%$. Los resultados obtenidos en las diferentes simulaciones realizadas demuestran que el cuadricóptero es capaz de detectar y seguir de forma autónoma un carro de radio control que se mueve en tierra.

Índice general

Índice de figuras

Índice de tablas

Lista de siglas

AMCL Adaptive Monte Carlo Localization

AMR Autonomous Mobile Robots

ANN Artificial Neural Network

AOI Automated Optical Inspection

API Application Programming Interface

CBIR Content Based Image Retrieval

CNN Convolutional Neural Networks

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DNNs Deep Neural Networks

DRAM Dynamic Random Access Memory

EDA Electronic Design Automation

GCS Ground Control Station

GPIO General Purpose Input/Output

GPS Global Positioning System

GPU Graphics Processing Unit

HIL Hardware-in-the-loop

IA Inteligencia Artificial

IMU Inertial Measurement Unit

IoT Internet of Things

LIDAR Laser Imaging Detection and Ranging

mAP mean Average Precision

NLP Natural Language Processing

NMS Non-maximum suppression

ODE Open Dynamics Engine

OGRE Object Oriented Graphics Rendering Engine

PID Proportional Integral Differential

PMIC Power Managment Integrated Circuits

R-CNN Region-based Convolutional Neural Networks

RoI Region of Interest

ROS Robot Operating System

RPN Region Proposal Network

SBC Single Board Computer

SDK Software Development Kit

SIFT Scale Invariant Feature Transform

SLAM Simultaneous Localization And Mapping

SOM System on Module

SSD Single Shot MultiBox Detector

SVM Support Vector Machines

UAV Unmanned Aerial Vehicle

UGV Unmanned Ground Vehicle

URDF Unified Robot Description Format

VGG Visual Geometry Group

VOC Visual Object Classes

XML Extensible Markup Language

YOLO You Only Look Once

1 Introducción

Los vehículos aéreos no tripulados (UAV) también llamados drones, son aeronaves que para ser pilotadas no requieren de la intervención de un piloto a bordo. Los UAV pueden ser controlados por teleoperación remota por una persona en una estación en tierra (*Ground Control Station (GCS)*) o también pueden volar de forma autónoma cuando se les equipa con diferentes dispositivos y sensores que les permitan realizar tareas programadas.

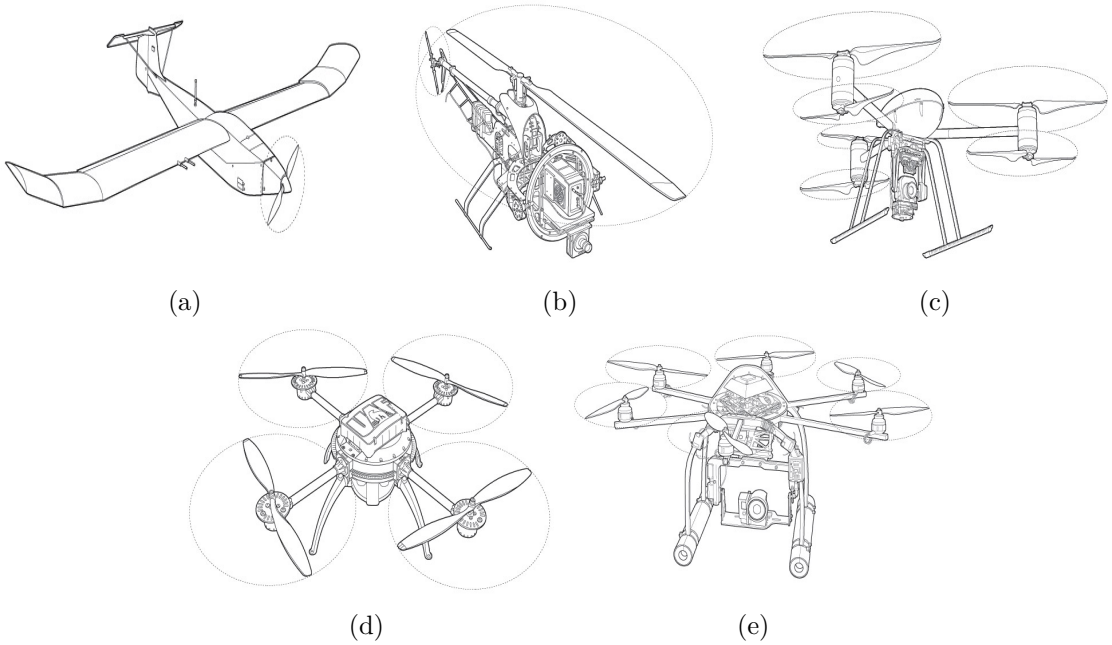


Figura 1.1: Modelos UAV: **a:** Ala fija, **b:** Helicóptero, **c:** Tricóptero, **d:** Cuadricóptero y **e:** Hexacóptero. Tomado de [1].

Existen muchas clasificaciones de UAV teniendo en cuenta características, como el peso, tamaño, altitud de operación, rango de misión, tiempo de operación, tipo de despegue, prin-

cipio de vuelo, cantidad de motores de operación, la capacidad y la combinación de estas. La Figura 1.1 muestra la clasificación más comúnmente usada de acuerdo con la cantidad de motores que utiliza la plataforma para el vuelo.

Con la tecnología actual y las diversas tareas de un UAV, se ha incrementado el desarrollo de estas plataformas aéreas, cada vez es mayor su auge y las diferentes aplicaciones en las que se encuentran. Algunas de las empresas más populares que fabrican y distribuyen drones son: DJI[1] con las series Phantom, Spark y Mavic; Parrot[2] con las series AR.Drone, Bebop y Anafi; y Skydio[3] con las series Skydio 2 y Skadio X2. Estas empresas buscan competir con el desarrollo de UAV que superen los niveles de autonomía y rendimiento de vuelo de modelos competidores en el mercado. Un aspecto importante de los UAV autónomos es el sistema de navegación que utilizan, en principio, implica tres tareas comunes, la primera es estimar la posición y orientación del UAV, la segunda es identificar los obstáculos y tomar decisiones de acuerdo a esto, y por último, enviar comandos para guiar y controlar el vuelo.

Actualmente, los drones son utilizados en sectores como el agropecuario, la industria, el transporte, el comercio, la construcción, la minería y las comunicaciones. Con tareas propias que implican la vigilancia, inspección y monitoreo, reconstrucción de escenas, filmaciones, transporte de productos, entre otros. En los UAV modernos se busca mayor autonomía, rendimiento en los tiempos de vuelo y que incluya un sistema de navegación robusto y confiable para realizar estas tareas.

1.1. Descripción del problema

Los UAV han demostrado ser útiles en diversos sectores de la industria donde se necesitan cumplir tareas de inspección, supervisión, vigilancia y exploración. Durante los últimos años, se ha incrementado el número de aplicaciones en las que se emplean los UAV, donde muchas de las actividades realizadas requieren de una vigilancia visual aérea. Actualmente, existen aplicaciones basadas en visión para los UAV, como por ejemplo, en la inspección de líneas de transmisión de alta tensión [2], vigilancia en áreas marítimas [3], detección de incendios forestales [4], búsqueda y rescate [5], entre otros. Sin embargo, la mayoría de estas aplicaciones están diseñadas considerando un operador humano como intermediario, que se encarga de controlar el vuelo y las maniobras que debe realizar el UAV. Este tipo de control de vuelo tiene como restricción el tiempo de operación y costos que conlleva disponer de un piloto que cuente con las habilidades técnicas para maniobrar un UAV de acuerdo con los requerimientos de la aplicación. Por lo anterior, se ha incrementado la necesidad de automatizar el control y aumentar las capacidades de navegación, con mejores niveles de robustez, precisión y autonomía de estas plataformas. Los sistemas basados en visión, que combinan algoritmos de detección y seguimiento de objetos son una fuente de información prometedora para el control de estos vehículos aéreos no tripulados.

En ambientes exteriores, uno de los sistemas de navegación más común para los UAV, se basa en la localización por GPS (*Global Positioning System (GPS)*) [6], [7]. Los drones

[1] https://www.dji.com/
[2] https://www.parrot.com
[3] https://www.skydio.com/

en este caso, utilizan los receptores GPS que proporcionan las coordenadas globales para el control y una navegación inteligente. La precisión de estos sistemas depende directamente del número de los satélites con los que el dispositivo se encuentre conectado, lo que representa una limitación para la navegación, en lugares donde la señal puede ser interferida o incluso completamente bloqueada como ocurre en entornos de GPS denegado (GPS-*denied*) [8].

Estas dificultades y necesidades, han permitido considerar la visión por computador y los sensores ópticos, como una de las principales alternativas de bajo costo para el control de la navegación de los UAV [9],[10], [11] gracias a su capacidad para capturar información detallada de los ambientes interiores y exteriores. La capacidad de detección y seguimiento visual desde los UAV, son tareas importantes que permiten completar con éxito las misiones que tienen habitualmente estas plataformas aéreas.

En el campo de la visión por computador, la detección y el seguimiento de objetos son fundamentales no solo para el desarrollo de aplicaciones UAV, sino también para el funcionamiento de muchas otras aplicaciones, como por ejemplo: la supervisión del tráfico [12], navegación autónoma de los robots [13], vigilancia de grandes edificaciones [14], realidad aumentada [15], [16], interacción humano-computador [17], seguimiento de personas [18], análisis estadísticos a deportistas [19] y vehículos autónomos [20]. Estas aplicaciones tienen en común tareas que requieren ubicar y hacer el seguimiento de objetos o regiones de interés en una secuencia de imágenes para luego ejecutar determinadas acciones.

Tanto la detección como el seguimiento de objetos son consideradas tareas complejas para los sistemas de visión por computador. Como se observa en la Figura 1.2, factores como la deformación del objeto, la complejidad geométrica del objeto o su entorno, la variación de escala, cambios de iluminación, oclusión parcial o completa, cambios de punto de vista y los movimientos rápidos de los objetos que pueden cambiar su apariencia, son algunos de los desafíos que continúan siendo objeto de investigación en la visión por computador. Estos desafíos y la importancia del seguimiento visual han incrementado la publicación de artículos y eventos científicos donde se presentan y evalúan el rendimiento de diferentes algoritmos rastreadores, como el evento VOT Challenge[4], donde la comunidad participa en desafíos y comparten los avances en el campo del seguimiento visual de objetos. La localización y detección de objetos, son tareas que se han convertido en temas de estudio y organización de numerosos eventos, conferencias, retos y talleres en torno a impulsar la investigación en el campo del seguimiento visual de objetos, como es el caso del evento organizado por ImageNet Challenge[5].

[4] http://www.votchallenge.net/
[5] http://image-net.org/

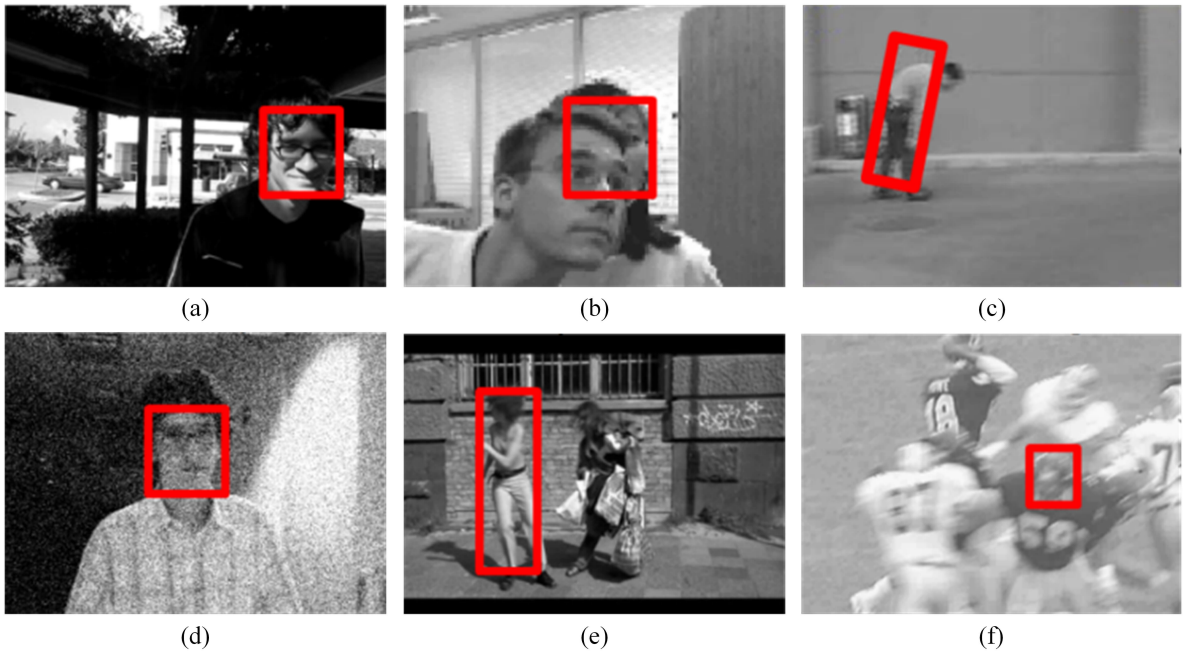


Figura 1.2: Desafíos del seguimiento visual de objetos: **a:** Iluminación, **b:** Oclusión, **c:** Deformación, **d:** Ruido, **e:** Rotación fuera del plano y **f:** Movimiento borroso. Tomado de [21].

Como se mencionó anteriormente, la tarea de seguimiento de objetos implica tener en cuenta múltiples variables al momento de desarrollar un algoritmo rastreador de objetos. El seguimiento visual de objetos implica tres etapas claves: la detección de objetos de interés que se encuentran en movimiento, el seguimiento visual de estos objetos en cada recuadro y una tercera etapa, que corresponde al análisis de secuencias de imágenes para la toma de decisiones. De acuerdo con la revisión bibliográfica, existen trabajos académicos que han abordado este problema, sin embargo, la mayoría de enfoques propuestos basan su funcionamiento en el envío de la señal de video del UAV a una estación terrestre, para desde allí realizar el procesamiento de imágenes sobre un computador [22].

Los algoritmos de visión presentados en este proyecto se implementaron sobre un ordenador de placa reducida que tiene conectada una cámara monocular para hacer el procesamiento de imágenes desde el UAV. Un sistema integrado a bordo del robot aéreo permite procesar y usar la información visual capturada de forma inmediata para dirigir directamente el cuadricóptero. El diseño de un sistema de visión embebido representa un reto en el uso de componentes *hardware*, y por consiguiente en recursos de procesador, memoria, velocidad de procesamiento y almacenamiento de datos, por lo que es necesario seleccionar componentes fáciles de transportar por el UAV.

El desarrollo de los algoritmos rastreadores, consideró aspectos importantes como la altura de operación del UAV, la velocidad a la que se mueve el objeto en tierra, el rendimiento de los algoritmos para rastrear el objeto móvil y la resolución de la imagen de entrada. En este caso, el algoritmo rastreador tiene como tareas principales: estimar la posición (x, y) del objeto en la secuencia de fotogramas capturados, calcular el ángulo (θ) y la distancia de error que

existe entre el objeto móvil y el centro de visión de la cámara, y finalmente, enviar valores de velocidad para reposicionar el cuadricóptero.

El uso de estas plataformas aéreas está beneficiando las actividades en diferentes sectores industriales y de servicios alrededor del mundo, por lo tanto, dotar un UAV con la capacidad de detección y seguimiento de determinados objetos móviles, abre un gran abanico de posibles aplicaciones relacionadas con la inspección, la vigilancia y el monitoreo de instalaciones y edificaciones. Este trabajo de grado aporta un punto de inicio para futuros desarrollos o trabajos académicos que involucren el seguimiento de objetos basados en visión por computador desde vehículos aéreos no tripulados usando Robot Operating System (ROS).

1.2. Contenido del documento

Este documento se encuentra organizado por capítulos de la siguiente manera:

Capítulo 2: Marco teórico
Se presenta una revisión del estado del arte, la literatura académica y artículos de investigación que se relacionan con este trabajo de grado. Se presenta el marco conceptual y las definiciones que buscan contextualizar al lector con la temática de este trabajo. Conceptos como visión por computador, *machine learning*, *deep learning*, redes neuronales convolucionales, algoritmos de detección de objetos, arquitectura de ROS y sistemas embebidos, entre otros.

Capítulo 3: Metodología
Este capítulo presenta la metodología utilizada y se describe cada etapa realizada para diseñar, desarrollar e implementar el sistema de visión por computador embebido para dotar a un UAV con la capacidad de seguir un objeto terrestre en movimiento. Se evidencia el porqué de la selección de los componentes *hardware* y *software* que integran el sistema, así como sus especificaciones técnicas.

Capítulo 4: Detección de objeto
En este capítulo se describe la estrategia usada desde la conformación del banco de imágenes *dataset* del objeto a detectar, la anotación de las imágenes, el pre-procesamiento de datos, hasta llegar al entrenamiento de los tres modelos *deep learning* presentados en este trabajo: SSD Inception v2, SSD MobileNet v1 y Faster R-CNN Inception v2.

Capítulo 5: Implementación
En este capítulo se presentan las etapas a seguir para el desafío de implementar los modelos *deep learning* sobre el módulo embebido Jetson TX2. En primer lugar, se muestra cómo se realiza la integración de Tensor-RT para conseguir optimizar las inferencias TensorFlow. Se comparan los modos NVPModel que incluye Jetson TX2 y se muestran resultados de cómo su uso permite aumentar el rendimiento en la ejecución de los modelos *deep learning*. Se presentan los cálculos para estimar la pose (x, y) y orientación (θ) del objetivo móvil terrestre, respecto al cuadricóptero. Así mismo, para la estimación de la distancia error y las velocidades lineales y angulares del cuadricóptero.

Capítulo 6: Interfaz de comunicación

En este capítulo se expone la importancia del uso de ROS en el desarrollo y despliegue de aplicaciones robóticas, y se describe la forma en que se implementa la interfaz de comunicación entre el módulo Jetson TX2 y el robot cuadricóptero. Además, contiene la descripción del entorno de simulación a través del uso de Rviz y Gazebo, con el fin de hacer un despliegue realista de la aplicación y comprobar su funcionamiento.

Capítulo 7: Resultados y contribución

En este capítulo se presentan los experimentos y las simulaciones de vuelo realizadas con el cuadricóptero, guiado de forma automática por el sistema de visión que detecta y sigue el objeto que se mueve en tierra. Adicional, se muestran los resultados obtenidos en el funcionamiento de los algoritmos y del sistema de visión, se incluye una comparación en el rendimiento y precisión de tres modelos *deep learning*. Así mismo, se describen aspectos de operabilidad del sistema, como la altitud máxima del cuadricóptero para detección y seguimiento del objeto móvil, y la velocidad a la que puede moverse.

Capítulo 8: Conclusiones

Este capítulo resume las conclusiones de los algoritmos y el sistema de visión embebido propuesto para el cuadricóptero. Se analizan las posibilidades de trabajos futuros.

2 Marco teórico

En este capítulo se presenta una revisión bibliográfica de las aplicaciones UAV, los algoritmos y las técnicas de visión por computador usados en la detección y el seguimiento de objetos. Además, se muestra un marco de conceptos y tecnologías que se relacionan con el desarrollo de este trabajo.

2.1. Estado del arte

Una plataforma aérea no tripulada (UAV) es aquella capaz de realizar una misión sin necesidad de tener una tripulación embarcada y con algún grado de autonomía. Cabe resaltar que ésta condición no excluye la existencia de piloto, controlador de la misión u otros operadores que pueden realizar su trabajo desde tierra [23].

Dentro de los UAV de tamaño pequeño, se encuentra el cuadricóptero que es un tipo de vehículo aéreo no tripulado que se caracteriza por tener con cuatro rotores que utiliza para sostenerse en el aire, despegar o aterrizar en forma vertical y pueden ser controlados para realizar tareas en pequeñas áreas, lo que representa una ventaja de operación con respecto a los UAV de ala fija. El cuadricóptero tiene seis grados de libertad: tres movimientos rotativos (*roll*, *pitch* y *yaw*), tres movimientos de traslación en los ejes (x, y, z). Un marco de coordenadas del cuerpo $\left[x^b, y^b, z^b\right]$, y un marco de coordenadas global $\left[X^G, Y^G, Z^G\right]$. *Roll*($\varphi$) representa la rotación sobre el *eje-x*, *pitch*(θ) es la rotación en el *eje-y*, y *yaw*(ψ) es la rotación en el *eje-z*. Cualquier orientación del cuadricóptero puede describirse mediante la combinación de estos ángulos. El sistema de coordenadas está dividido en un marco de referencia global $\{G\}$ y un marco de referencia del cuerpo $\{b\}$ como se observa en la Figura 2.1.

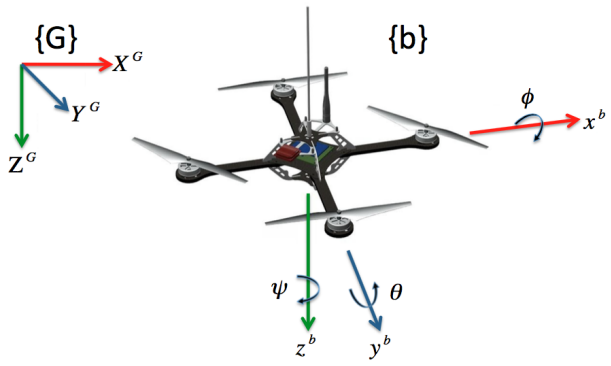


Figura 2.1: Esquema de cuadricóptero UAV. Tomado de [24].

Las primeras aplicaciones reales de la tecnología de los vehículos aéreos no tripulados fueron utilizadas con fines militares [25] y consistían en aviones pilotados de forma remota. Sin embargo, en la actualidad se han desarrollado un gran número de aplicaciones civiles que incluye el control de tráfico [26], topografía [27], estudio de los suelos [28], exploración de petróleo y gas, reconstrucción de planos [29], recolección de datos para aplicaciones meteorológicas [30], ubicación de incendios [31], agricultura de precisión [32], control de cosechas [33], inspección de líneas eléctricas de alto voltaje [34], comunicaciones móviles e internet de la cosas [35] y se siguen explorando aplicaciones como por ejemplo el transporte de mercancías [36], filmaciones de juegos deportivos [37] o eventos sociales, por nombrar algunas de las más importantes.

(a) (b) (c)

(d) (e) (f)

Figura 2.2: Aplicaciones de los UAV: **a:** Agricultura de precisión. Tomado de [38]. **b:** Estadísticas deportistas. Tomado de [39]. **c:** Entretenimiento. Tomado de [40]. **d:** Rescate. Tomado de [41]. **e:** Inspección de líneas eléctricas de alta tensión. Tomado de [42]. **f:** Transporte de mercancía. Tomado de [43].

Los UAV dentro de la industria aeronáutica son una de las áreas con mayor crecimiento [44], lo que se demuestra en el hecho de que su uso se ha multiplicado en la última década. En comparación con los vehículos aéreos tripulados, estas aeronaves se pueden maniobrar más fácilmente y sus costos de uso pueden llegar a ser muy inferiores. Con ellas se puede evitar el riesgo inherente de los vuelos tripulados en entornos hostiles, en condiciones de vuelo con escasa visibilidad o en condiciones climatológicas adversas.

Un UAV requiere una infraestructura especial, razón por la que también se habla de sistemas aéreos no tripulados. Estos sistemas suelen estar constituidos principalmente por la plataforma aérea y por dispositivos en tierra que incluye el sistema de control del vehículo aéreo, los equipos de comunicaciones, la estación de mando, control y comunicación [44].

2.1.1. Aplicaciones autónomas basadas en visión para UAV

La visión por computador y el procesamiento de imágenes se han convertido en herramientas fundamentales para el desarrollo de diferentes tareas complejas que realizan los UAV. En la actualidad se busca que las aplicaciones para los UAV sean autónomas para que realicen acciones como el despegue, aterrizaje, seguimiento de objetos, vigilancia, exploración e inspección de aéreas. A continuación, se presentan varios trabajos académicos y de investigación que proponen soluciones ante los desafíos que implican las aplicaciones basadas en visión por computador como se muestra en la Tabla 2.1. La mayoría de estas aplicaciones realizan el procesamiento de imágenes en una estación ubicada en tierra (*Ground Control Station (GCS)*).

Aplicación	**Descripción**	**Técnicas de visión**	**Objeto detectado**	**Trabajos relacionados**
Aterrizaje autónomo	Despegue y aterrizaje autónomo	AprilTag 'ar track alvar'	Códigos QR, marcas	[45, 46, 47, 48] [49, 50]
Seguimiento de objetos	Algoritmos de extracción de características	SIFT, SURF, ORB, RANSAC	plantillas, carro, persona	[51, 52, 53, 54] [55, 56, 57, 58]
	Detectores de objetos *Deep Learning*	CNN, SSD, Faster R-CNN	carros, UAV personas objetos	[59, 60, 61, 62] [63, 64, 65, 66]

Tabla 2.1: Trabajos relacionados y aplicaciones UAV autónomas basadas en visión por computador.

2.1.2. Despegue y aterrizaje autónomo de UAV

El despegue y el aterrizaje autónomo son tareas fundamentales para los UAV. En [45] se presenta una revisión de técnicas de aterrizaje que hacen uso de GPS y de técnicas basadas en visión. Tiene como objetivo proporcionar perspectiva sobre el estado del problema de control de aterrizaje y diseño del controlador. En [47], se presenta un sistema de control difuso para el aterrizaje y control de altitud de un UAV que sobrevuela un objetivo móvil terrestre. La simulación de los controladores se validó con la combinación de herramientas de

código abierto FuzzyLite, `'ar_track_alvar'` y ROS Gazebo con `tum_simulador` de ArDrone 2.0. Los resultados de la simulación mostraron aceptables rendimiento de los controladores de altitud y aterrizaje.

En [49], se presenta un algoritmo robusto basado en visión para el seguimiento de marcadores y aterrizaje autónomo de un UAV. Para distancias cortas se adoptan círculos concéntricos como marcador. Para aumentar el rendimiento y disminuir el costo del sistema, se usaron sensores de visión en lugar de GPS o sensores láser. Con el desarrollo de tecnología *hardware* para el procesamiento de imágenes, se obtuvo información precisa para la navegación y el aterrizaje autónomo. Los resultados de las simulaciones mostraron que el algoritmo funciona con mayor precisión si se fusiona con la información provista por una IMU y un altímetro.

En [50], se propone una solución para aterrizar un UAV sobre un vehículo terrestre no tripulado (*Unmanned Ground Vehicle (UGV)*) que se desplaza en un ambiente exterior. El control propuesto basa su funcionamiento en el procesamiento de imágenes, junto con un sistema de navegación por GPS. La posición del UAV se estima utilizando los datos del GPS, mientras que la distancia entre la pista de aterrizaje y el UAV se determina en función de las imágenes procesadas. La plataforma de aterrizaje se compone de dos partes; un patrón de $0.6m \times 0.6m$ que pueda ser detectado desde grandes altitudes, y un código QR de $0.2m \times 0.2m$ para guiar el cuadricóptero cuando sobrevuela cerca de la plataforma de aterrizaje. Todos los experimentos se realizaron en un ambiente de simulación usando ROS y V-REP.

2.1.3. Seguimiento de objetos autónomo desde UAV

En [51], se estudia un algoritmo para el seguimiento de objetos terrestres desde un UAV. Se trata del algoritmo *Scale Invariant Feature Transform (SIFT)*, que tiene un buen rendimiento ante rotaciones, cambios en la escala y la iluminación de la imagen. Se usa para extraer y comparar características con el fin de reconocer objetos. La anterior función la combinan con el filtro de Kalman para estimar la posición del objetivo en el siguiente fotograma, mejorar la rapidez del seguimiento y disminuir las probabilidades de perder el objetivo. Realizan un montaje experimental para probar el algoritmo desarrollado, que consta de un UAV, una cámara que transmite la señal de forma inalámbrica a una computadora en tierra. Sobre esta computadora se realiza el procesamiento de imágenes, utilizando la biblioteca de OpenCV y programación en C++.

En el trabajo [67] se propone una estrategia de estabilización de UAV basada en visión por computador y controladores de conmutación, con el objetivo de realizar un sistema que realice el seguimiento de objetos móviles en el suelo. La arquitectura *hardware* consiste en equipar a un cuadricóptero con una cámara integrada que envía la fuente de imágenes capturadas a una estación en tierra, donde una computadora junto con un algoritmo de visión por computador se encarga de procesar las imágenes. Proponen un estimador basado en visión, donde a partir de imágenes bidimensionales calculan la posición tridimensional y la velocidad de traslación del UAV respecto al objeto móvil. Presenta estrategias de control que permiten buscar el objeto cuando está fuera del campo de visión de la cámara o cuando se pierde de manera temporal.

El proyecto [68] considera como problema de estudio, encontrar un método eficiente para

el seguimiento de objetos con un UAV. Usan el AR.Drone 2.0 como plataforma aérea para rastrear el objetivo móvil y ROS para el control del dron y el seguimiento de una etiqueta de realidad aumentada (AR). El paquete de ROS `'ar_track_alvar'` permite generar etiquetas AR, identificar y hacer el seguimiento de las mismas. Usan el paquete ROS `ardrone\_autonomy'` para establecer la comunicación vía WiFi entre el dron y una computadora que corre Ubuntu y ROS. Para realizar el seguimiento de la etiqueta, implementan un controlador difuso, junto con reglas de lógica difusa definidas teniendo en cuenta las entradas del sistema y las salidas esperadas.

En [69], se describe un sistema de visión para un cuadricóptero autónomo. El control se basa en un algoritmo de detección y seguimiento de objetivos móviles de determinado color, se utilizan filtros de Kalman para la estimación relativa de postura y un controlador no lineal para la estabilización y orientación del UAV. El sistema tiene una estación en tierra donde se hace el procesamiento de imágenes y se puede elegir un objetivo de color arbitrario para su seguimiento.

En [70], se propone un algoritmo basado visión, con el fin de rastrear y perseguir de forma autónoma un objetivo en movimiento con un pequeño UAV. El sistema desarrollado para el seguimiento basa su funcionamiento de acuerdo al color del objeto y se encuentra en una estación en tierra, permite manejar grandes desplazamientos y oclusiones temporales del objetivo.

La investigación [71] se enfoca en conseguir que un UAV detecte objetos en movimiento en medio de los océanos. Por lo que desarrollan un algoritmo para rastrear, hacer el seguimiento y aterrizar el cuadricóptero sobre marca puesta en un *kayak*. Con la ayuda de MATLAB se desarrolló el algoritmo de control, la corrección de la cámara, corrección de inclinación y la corrección de brillo. Estos algoritmos tienen en cuenta las difíciles condiciones ambientales en los océanos debido a los vientos y las corrientes que hacen que el *kayak* esté en movimiento. Los resultados obtenidos en la investigación arrojaron un porcentaje de éxito de 75-80 %.

En [72], se utilizó ROS para desarrollar algoritmos de generación y seguimiento de trayectorias para el cuadricóptero AR.Drone 2. Para crear las rutas de vuelo se usaron polinomios cúbicos y curvas de Bézier. Un controlador PID (*Proportional Integral Differential (PID)*) para conducir el dron desde su posición actual a la coordenada de punto de partida. Se implementó un método de navegación basado en campos potenciales para evadir obstáculos y alcanzar objetivos. Se realizó la simulación en Gazebo con un modelo del dron y también pruebas reales con el AR.Drone 2.0.

2.2. Visión por computador

La visión por computador inició con el desarrollo de técnicas matemáticas para recuperar la forma tridimensional y la apariencia de los objetos en las imágenes. En general, la visión por computador busca desarrollar métodos que sean capaces de replicar el funcionamiento del sistema visual humano. Cualquier aplicación de visión por computador busca cumplir con dos tareas principales, la de percibir su entorno y la segunda, tomar acciones basadas en estas percepciones [73], [74].

En la actualidad existen técnicas para computar con precisión imágenes digitales, de modo que las máquinas puedan extraer información, resolver alguna tarea, reconstruir una escena y objetos que están visionando. Sin embargo, estos avances todavía no permiten que un computador interprete una imagen al mismo nivel que una persona. Hacer que un computador procese una imagen es completamente diferente de entender lo que está sucediendo en la imagen, lo que no es una tarea sencilla. Por lo que el procesamiento de imágenes representa una pieza dentro de un sistema mucho más complejo que tiene como objetivo interpretar el contenido de la imagen [73].

La comunidad de investigación en este campo ha contado con el apoyo de grandes empresas TI que utilizan sistemas de visión por computador para el desarrollo de sus productos o implementación de sus servicios. Gracias a la disponibilidad de los datos y los sensores visuales, cada vez más se están abordando problemáticas con mayores desafíos en este campo, por ejemplo, en la navegación de los vehículos autónomos basados, en la recuperación o consulta de imagen basada en contenido (*Content Based Image Retrieval (CBIR)*) y en teléfonos inteligentes, entre otros.

El objetivo principal de la visión por computador es interpretar y dar sentido las imágenes, es decir, extraer información visual significativa o también información de alto nivel (semántica). Por ejemplo, identificar los objetos presentes en las imágenes, determinar su ubicación y cantidad. Algunos de los principales problemas que aborda la visión por computador se encuentran: la clasificación de objetos o de imágenes, la identificación de objetos, la detección y localización de objetos, segmentación e instanciación de objetos, estimación de pose y el seguimiento de objetos [75].

Clasificación de objetos: consiste en asignar las etiquetas apropiadas (clases) a un conjunto de imágenes. Esta tarea fue una de las primeras en las que se usó de forma exitosa las redes neuronales convolucionales (*Convolutional Neural Networks (CNN)*).

Identificación de objetos: la tarea de identificar objetos son métodos que se encargan de aprender a reconocer instancias específicas de una clase.

Detección y localización de objetos: su función es determinar dónde están ubicados objetos específicos en una imagen. Se suele emplear en la detección de rostros para aplicaciones de vigilancia, en la revisión de radiografías en la medicina y en la detección de componentes averiados en plantas industriales.

Segmentación e instanciación de objetos: la segmentación es un tipo de detección, pero más avanzada. En lugar de proporcionar un cuadro delimitador de los objetos reconocidos, los métodos de segmentación retornan máscaras etiquetadas de todos los pixeles que pertenecen a una clase o instancia específica de una clase. Esta tarea se utiliza en aplicaciones robóticas y vehículos inteligentes, con el objetivo de entender su entorno.

Estimación de pose: para objetos rígidos esta tarea se encarga de estimar las posiciones y orientaciones de objetos con respecto a la cámara en el espacio 3D, en función de la representación 2D en una imagen. Esta es útil para que robots puedan interactuar con el entorno por ejemplo, para mover objetos o evitar colisiones. También se utiliza en aplicaciones de realidad aumentada para la superposición de contenido multimedia sobre

los objetos. En el caso de los objetos no rígidos, la estimación de pose puede significar la estimación de sus partes, por ejemplo, si se considera una persona como un objeto no rígido, las aplicaciones típicas tienen que ver con el reconocimiento de la posturas o acciones humanas como estar de pie, estar sentada, correr, entre otras.

Seguimiento de objetos: esta tarea se aplica en flujos de video, donde se requiere localizar objetos específicos en una secuencia de imágenes. Existen diferentes métodos utilizados para la detección e identificación de los objetos en cada trama. Una aplicación típica es el modo de seguimiento (*active track*) con el que cuentan algunos drones en el mercado, estos modos requieren la selección previa del área del objeto que se quiere seguir, esta acción se realiza a través de la aplicación para dispositivos móviles que provee la marca del dron.

2.3. Machine Learning

Machine learning es considerado como un subcampo de la inteligencia artificial (IA), así como el *deep learning* un subcampo del *machine learning* tal como se representa en la Figura 2.3. El objetivo principal de la IA es proveer un conjunto de técnicas que se puedan utilizar para resolver problemas que los humanos realizan casi de forma intuitiva y automática, pero que representan un desafío para un computador [76].

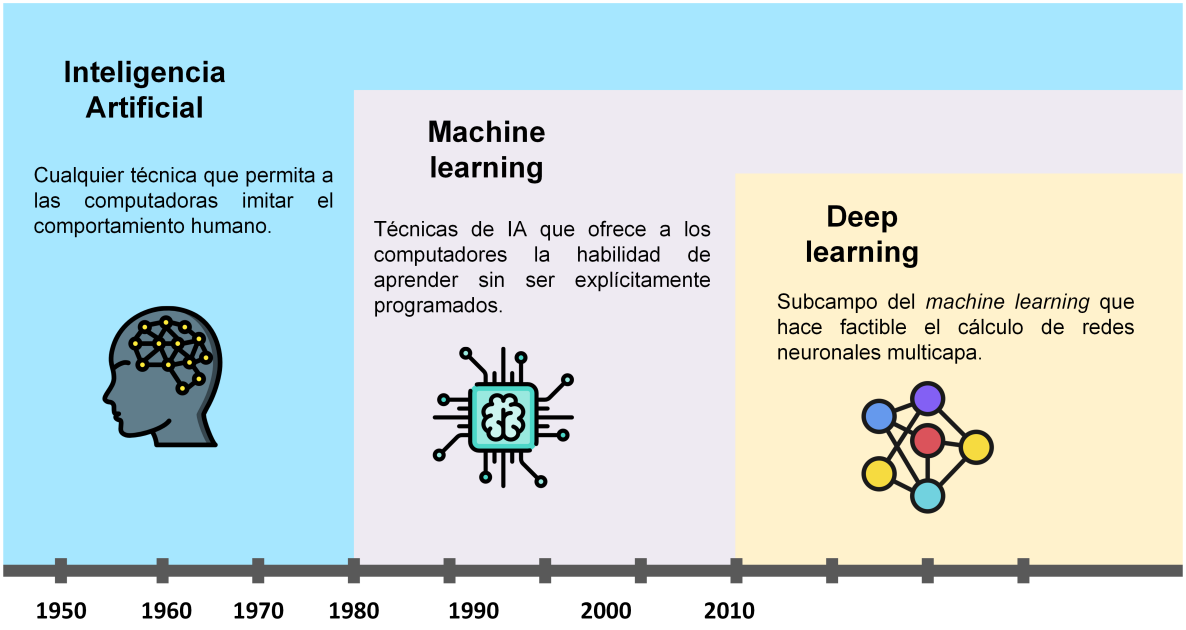


Figura 2.3: Línea de tiempo de la inteligencia artificial, *machine learning* y *deep learning*.

El aprendizaje automático o *machine learning* permite a los computadores aprender a partir de datos sin la necesidad de ser programados de forma explícita [73]. Su objetivo principal es diseñar métodos que automáticamente aprendan usando observaciones del mundo

real o también llamados datos de entrenamiento, sin que se requiera la definición explícita de las reglas o lógica de una persona. Estos métodos se pueden dividir en tres principales categorías: aprendizaje supervisado, aprendizaje no supervisado y el aprendizaje por refuerzo [77].

Aprendizaje supervisado: como primera rama del *machine learning*, este aprendizaje se enfoca en aprender patrones a través de la conexión de las variables con los resultados conocidos y trabaja con los conjuntos de datos etiquetados. El algoritmo de aprendizaje se entrena con un conjunto de datos etiquetados que incluyen la solución deseada, este conjunto de datos actúa como maestro y se encarga de entrenar al modelo o máquina. Una vez entrenado el modelo, el algoritmo estará en la capacidad de descifrar los patrones existentes en los datos y de crear un modelo que pueda reproducir las mismas reglas con nuevos datos suministrados.

Aprendizaje no supervisado: a diferencia del aprendizaje supervisado, este aprendizaje no presenta datos previamente etiquetados. En cambio, el algoritmo recibe una gran cantidad de datos y las herramientas para comprender los datos. A partir de ahí, el algoritmo extrae información relevante y aprende a agrupar y organizar los datos de tal forma que puedan tener sentido para una persona u otro algoritmo. Esto lo realiza sin la necesidad de conocer las variables de salida y a través de la exploración de los datos sin etiquetar.

Aprendizaje por refuerzo: el aprendizaje por refuerzo es bastante diferente del aprendizaje supervisado y el no supervisado. Este método utiliza las observaciones recopiladas de la interacción con el entorno y prueba diferentes posibilidades hasta obtener la respuesta correcta. El algoritmo de aprendizaje por refuerzo es recompensado o penalizado con base en su respuesta, una respuesta acertada recibe la recompensa y una respuesta incorrecta es penalizada. Después de algunas iteraciones, este algoritmo buscará tomar decisiones que maximicen la recompensa y minimicen el riesgo.

2.4. Deep Learning

El *deep learning* permite estructurar algoritmos que trabajan con múltiples capas con el propósito de generar una red neuronal artificial que puede aprender y tomar decisiones inteligentes por sí misma. Las redes neuronales artificiales se componen de múltiples entradas, salidas y capas ocultas. A su vez, cada capa está conformada por unidades de neuronas artificiales que se encargan de transformar los datos de entrada en información que la siguiente capa puede usar para una determinada tarea predictiva. El término red neuronal artificial está inspirado en la red neuronal biológica de una persona, intenta imitar cómo la información es procesada en el cerebro humano, cuando las señales de entrada se activan lo suficiente, cada neurona artificial dispara una señal a todas las neuronas con las que se encuentra conectada. Gracias a esta estructura y a la interacción de millones de neuronas artificiales apiladas en capas, se produce una conducta de aprendizaje que permite a un modelo o una máquina aprender a través de su propio procesamiento de datos [74].

2.5. Redes Neuronales Convolucionales (CNN)

Red Neuronal Convolucional (CNN) es el tipo de arquitectura de redes neuronales más comúnmente usada en el aprendizaje profundo o *deep learning*, especialmente para datos multidimensionales como imágenes o videos. Una CNN funciona de forma similar a una red neuronal artificial (*Artificial Neural Network (ANN)*), a excepción de las primeras capas convolucionales. Las primeras capas convolucionales aprenden características básicas de bajo nivel, como por ejemplo, líneas y bordes, las capas intermedias aprenden características un poco más complejas como círculos, cuadrados, entre otras formas. Las capas finales, se encargan encontrar las características más complejas que permiten realizar tareas de clasificación, como por ejemplo identificar un objeto, partes del rostro humano o una persona [74]. Las CNN son ampliamente utilizadas en el campo de la visión por computador, en tareas como la clasificación y detección de objetos [78], [79], [80], [81], y también, en el reconocimiento de voz [82].

Para imágenes, una CNN toma como entrada un dato tridimensional ($altura \times ancho \times profundidad$). Existen diferentes tipos de capas que agrupadas de determinada manera permiten la construcción de una CNN. La configuración más común, está compuesta por una serie de capas agrupadas en las que se encuentran, las convolucionales [`CONV`], las de activación [`ACT` o `RELU`], las de reducción o *pooling* [`POOL`], las completamente conectadas [`FC`] y, finalmente, un clasificador [`SOFTMAX`] para obtener en la salida las probabilidades de clasificación. En texto se puede representar como: `INPUT` ⇒ `CONV` ⇒ `RELU` ⇒ `POOL` ⇒ `CONV` ⇒ `RELU` ⇒ `POOL` ⇒ `FC` ⇒ `SOFTMAX` [76]. En la Figura 2.4 se muestra un diagrama de una CNN con las capas que la conforman. Cada una de estas capas realiza operaciones que modifican los datos con la intención de aprender características específicas de los datos.

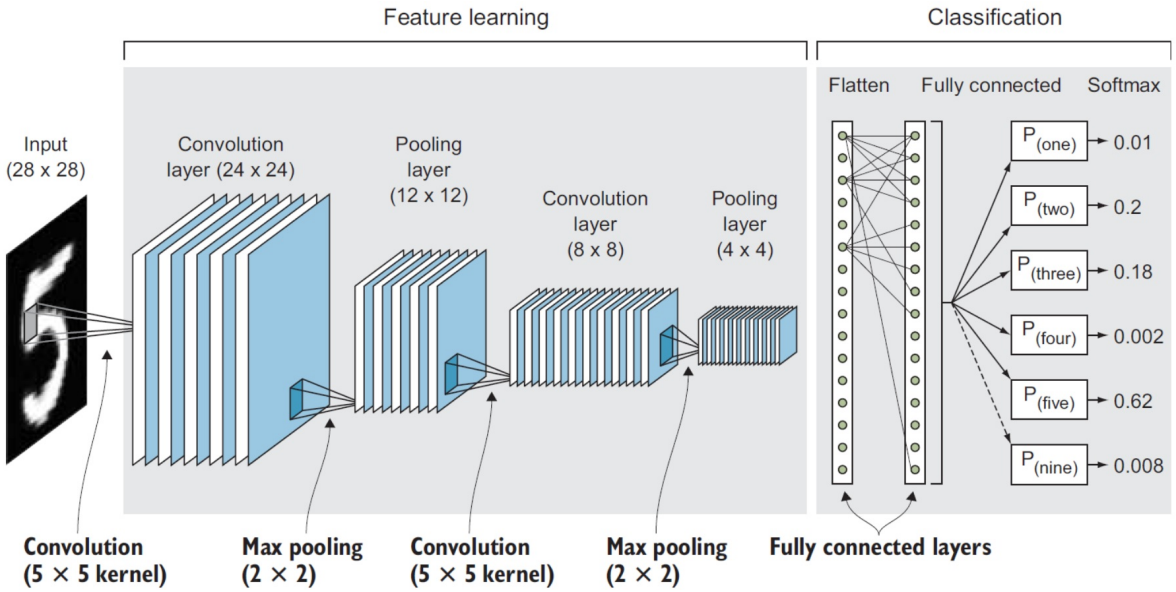


Figura 2.4: Representación de la arquitectura de una CNN. Tomado de [74].

2.6. Modelos detectores de objetos

La detección de objetos es uno de los problemas que aborda el campo de la visión por computador. Este problema involucra dos procesos principales, el primero consiste en localizar uno o varios objetos dentro que contiene una imagen y el segundo corresponde con la clasificación de cada objeto presente en la imagen y su clase. La salida de estas tareas se representa a través de un cuadro delimitador *bounding box* alrededor del objeto detectado. La entrada común en un algoritmo de detección de objetos es una imagen, mientras que la salida es un conjunto de *bounding box* y la clase del objeto correspondiente.

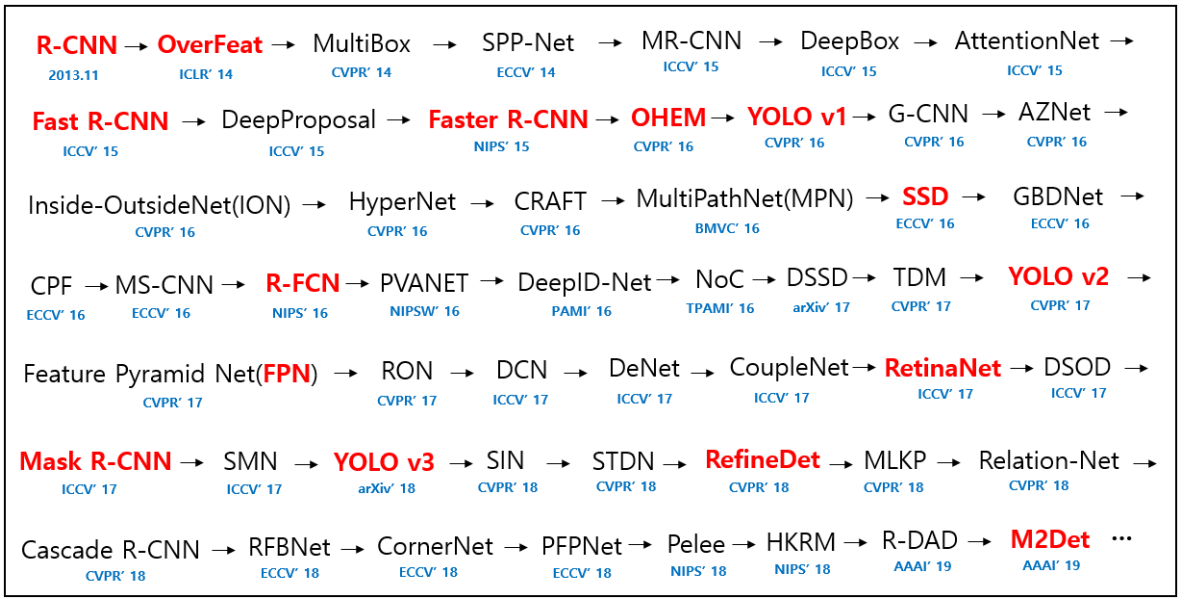


Figura 2.5: Estado del arte de los detectores de objetos *deep learning* en el periodo de 2013-2019. Los detectores resaltados (rojo), son considerados como importantes y más populares dentro de la literatura. Tomado de [83].

Los detectores de objetos modernos de aprendizaje profundo basan su funcionamiento en las redes neuronales convolucionales CNN. Algunos de los detectores de objetos más populares son, Region-based Convolutional Neural Networks (R-CNN), Faster R-CNN, Single Shot MultiBox Detector (SSD) y You Only Look Once (YOLO). La Figura 2.5 muestra una línea de tiempo con el estado de arte de los detectores de objetos *deep learning* con mayor relevancia en el periodo de 2103-2019. Un detector de objetos consta de cuatro elementos principales:

Propuesta de región: en este paso, el modelo o algoritmo acepta como entrada una imagen a la que se le ha aplicado varias capas convolucionales y como salida se obtienen regiones de interés Region of Interest (RoI). Las RoI, son las áreas o regiones que el sistema cree que podría contener un objeto y por consiguiente, asigna a los *boundig box* una puntuación o probabilidad cada cuadro delimitador. Las regiones con alta puntuación pasan a las siguientes etapas y no son tenidas en cuenta las regiones con baja puntuación.

Extracción de características y predicciones: en esta etapa, se analizan y se extraen

características de cada uno de los *bounding box* con alta probabilidad de contener un objeto identificados en el paso anterior. Estas regiones se evalúan para determinar cuáles objetos están presentes. Dos tipos de predicciones son realizadas en esta etapa, las coordenadas de cada *bounding box* y la predicción de la clase para cada objeto.

Supresión no máxima (SVM): uno de los problemas de los algoritmos detectores de objetos son las múltiples detecciones y *bounding box* que se generan de un mismo objeto. La técnica Non-maximum suppression (NMS) se encarga de analizar todos los *bounding box* que rodean un objeto, selecciona el cuadro delimitador que tiene la máxima probabilidad de predicción y elimina los demás recuadros con menor probabilidad.

2.6.1. Red Neuronal Convolucional basada en Región (R-CNN)

R-CNN inició en el año 2014 [84], este estudio propuso R-CNN como un algoritmo simple y escalable que combina dos enfoques principales, se pueden aplicar CNN a propuestas de regiones para localizar y segmentar objetos.

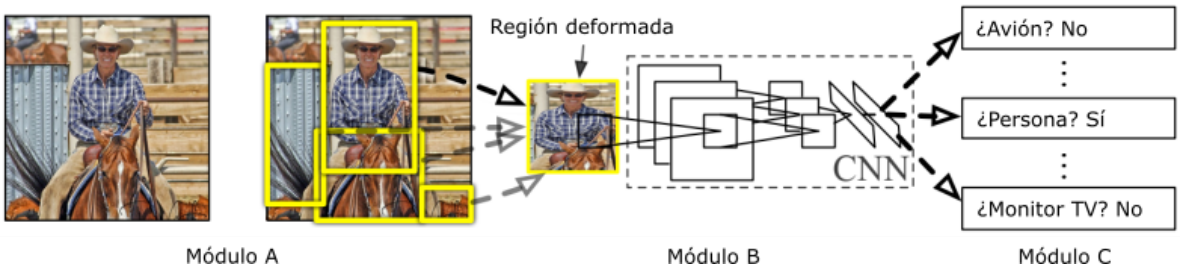


Figura 2.6: Modelo de arquitectura de R-CNN. **Módulo A:** imagen de entrada a R-CNN. Extracción de propuestas de regiones (RPN). **Módulo B:** cálculo y extracción de características para cada propuesta de región usando CNN. **Módulo C:** clasificación de cada región usando máquinas de soporte vectorial (SVM). Tomado de [84].

El modelo R-CNN se compone de tres elementos o módulos, como se observa en la Figura 2.6. El primer módulo representa la entrada de la imagen, de la que se extraen regiones de interés RoI, también llamadas propuestas de región. Para encontrar las regiones que tienen una alta probabilidad de contener un objeto se utiliza un algoritmo denominado *Selective Search*. Las propuestas de región se deforman, ya que las CNN requieren un tamaño fijo en la imagen de entrada. Luego se ejecuta una red de convolución pre-entrenada a las propuestas de región para extraer las características de una región candidata. El módulo final, con base en las características derivadas de cada CNN, tiene como función implementar múltiples máquinas de soporte vectorial (*Support Vector Machines (SVM)*) lineales aplicados a las detecciones candidatas encontradas en el paso anterior.

R-CNN ha representado un punto de partida en el desarrollo de otros detectores de objetos, sin embargo, tiene algunas desventajas con respecto a otros detectores más modernos. Un inconveniente radica en la lentitud del detector, esto se debe a que el algoritmo *Selective Search* identifica cerca de 2000 regiones de interés para una sola imagen que pasan a ser evaluadas en las otras etapas del modelo, lo que conlleva un alto costo computacional. Este

enfoque no es una buena opción, especialmente para aquellas aplicaciones en tiempo real que requieren ejecutar inferencias de manera muy rápida.

2.6.2. Fast R-CNN

Fast R-CNN es una versión mejorada para R-CNN [85]. Surgió en el año 2015 y aunque basa su funcionamiento en R-CNN, introdujo mejoras tanto en la velocidad, como en la precisión de detección de objetos. Para conseguir esto, este modelo incorpora dos cambios principales. A diferencia de R-CNN, Fast R-CNN realiza el entrenamiento de una sola CNN para una imagen de entrada, en lugar de entrenar múltiples CNN para todas las propuestas de región generadas para una imagen de entrada. Las múltiples máquinas de soporte vectorial (SVMs) también son reemplazados por un solo clasificador *softmax*. Los anteriores factores hacen que la etapa de entrenamiento sea más rápida en comparación con R-CNN (ver Figura 2.7).

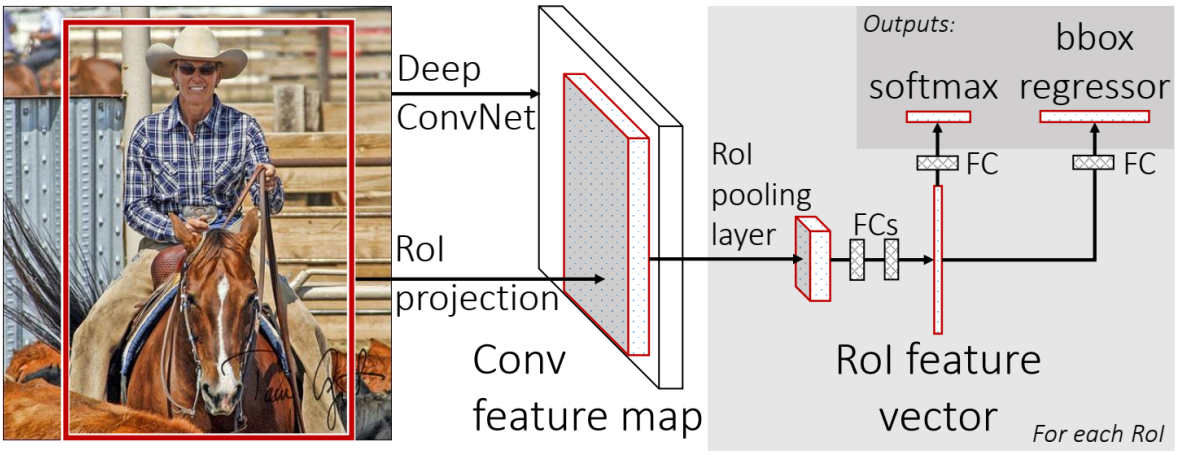


Figura 2.7: Modelo de arquitectura de Fast R-CNN. Una imagen de entrada y múltiples regiones de interés RoI son la entrada a una convolución completa. Tomado de [85].

La arquitectura de Fast R-CNN se representa en la Figura 2.7. Su funcionamiento consiste, en una imagen como entrada para una sola CNN con múltiples capas convolucionales para generar un mapa de características de convolución. Al igual que en R-CNN, las regiones de interés (RoI) son generadas con el algoritmo de búsqueda selectiva *Selective Search* y este a su vez, proporciona una entrada a Fast-RCNN. Cada una de las propuestas de región está deformada y se introduce en una capa *pooling* RoI. La capa *pooling* RoI produce un vector de características de longitud fija desde el mapa de características para cada uno de los objetos candidatos y luego lo pasa como una entrada a una serie de capas completamente conectadas [FC]. Las capas totalmente conectadas se dividen en dos ramas: el clasificador *softmax* que predice la clase de propuesta de región y la salida de la regresión proporciona los cuatro valores de desplazamiento para ajustar los cuadros delimitadores.

2.6.3. Faster R-CNN

A pesar de los avances que tuvo el modelo Fast R-CNN, todavía se tenía como requisito utilizar el algoritmo de búsqueda selectiva *Selective Search*. Aunque este algoritmo genera propuestas de región, representa el principal cuello de botella de estas redes, a causa de su lentitud y alto costo computacional. Por lo tanto, en 2015 los autores de Faster R-CNN [86], sustituyeron la búsqueda de selectiva por una red de propuestas de región (Region Proposal Network (RPN)) en un algoritmo de detección de objetos conocida como Faster R-CNN. La red propuesta de región utiliza una red profunda completamente convolucional para generar un número fijo de propuestas de región como se muestra en la Figura 2.8. Faster R-CNN unifica RPN con Fast R-CNN para detectar objetos.

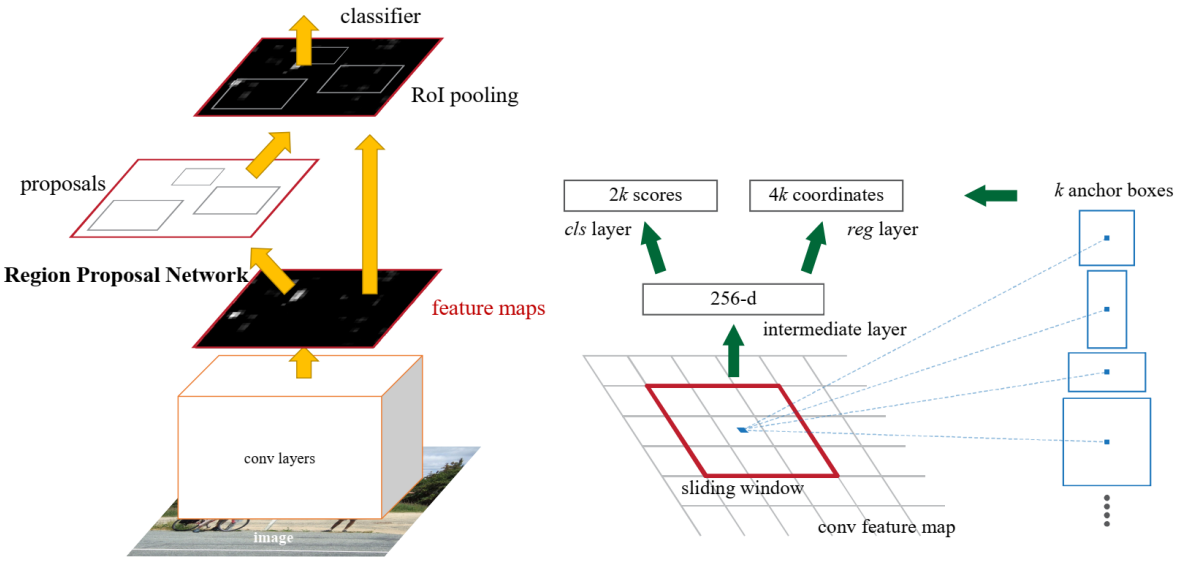


Figura 2.8: Modelo de arquitectura de Faster R-CNN. Tomado de [86].

2.6.4. Single Shot MultiBox Detector (SSD)

El detector de objetos de disparo único (*Single Shot MultiBox Detector (SSD)*) fue lanzado a finales del 2016 propuesto por Liu et al. [87]. A diferencia de los detectores de la familia R-CNN que son multietapa, los detectores SSD usan una sola red neuronal profunda, de allí se deriva su nombre. Este algoritmo predice el *bounding box* y la clase simultáneamente en un solo disparo. Realizar estas operaciones en un solo paso hace que SSD sea más rápido y una opción viable para las detecciones en tiempo real.

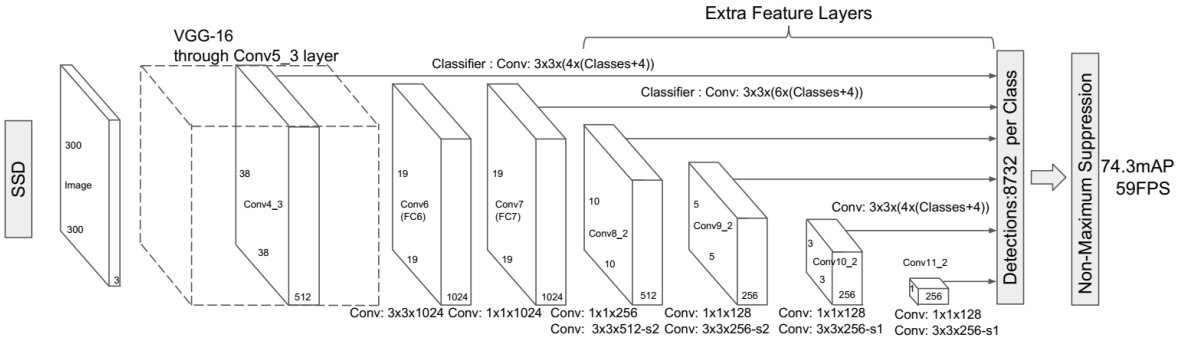


Figura 2.9: Modelo de arquitectura de SSD. Tomado de [87].

La Figura 2.9 describe la arquitectura de SSD que basa su funcionamiento en una red VGG16 (*Visual Geometry Group (VGG)*) [88]. Este es un modelo de red neuronal convolucional (CNN) presentado en la competencia ILSVR(Imagenet) en 2014[6]. El equipo participante obtuvo importantes resultados que los llevó a ocupar el primer y segundo lugar, en las tareas de localización y clasificación de objetos respectivamente. Es considerada una de las mejores arquitecturas de modelos de visión. Lo más singular de VGG16 es que, en lugar de tener una gran cantidad de hiperparámetros, se centraron en tener capas de convolución de filtro 3×3 con un paso de 1 y siempre usaron el mismo relleno y capa *maxpool* del filtro 2×2 en paso de 2. Sigue esta disposición de convolución y capas de *maxpool* consistentemente en toda la arquitectura.

Al final, tiene dos capas completamente conectadas [FC], seguido de un clasificador *softmax* para salida. El VGG16 cuenta con 16 capas que tienen pesos. Esta red es una red bastante grande y tiene aproximadamente 138 millones de parámetros. Después de la red base VGG16, se añade un conjunto de filtros de convolución. Estas capas disminuyen su tamaño progresivamente para permitir detecciones en múltiples escalas. Finalmente, se aplica supresión no máxima (*SVM*) para eliminar los *bounding box* que se sobreponen y mantener un solo recuadro para detectar cada objeto.

2.7. ROS (*Robot Operating System*)

ROS combina funcionales de *middelware* y de *framework* para el desarrollo de aplicaciones robóticas. ROS proporciona un ambiente de desarrollo rápido, con un amplio conjunto de herramientas para configurar, escribir y depurar algoritmos responsables del control y la navegación de plataformas robóticas (ver Anexo B). Dentro de los muchos paquetes disponibles en la comunidad ROS, el metapaquete *hector quadrotor* provee un modelo 3D para el control y la simulación de un cuadricóptero UAV. Las funcionalidades de ROS permiten la creación de nodos que se comunican entre sí mediante el intercambio de mensajes.

[6] http://www.image-net.org/challenges/LSVRC/2014/results

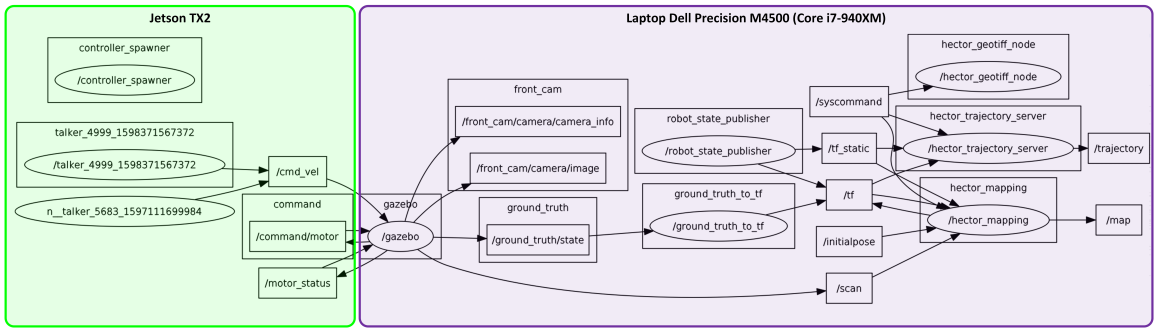


Figura 2.10: Gráfica de nodos ROS del sistema. **Izquierda:** nodos y *topics* ROS que se ejecutan en Jetson TX2 (recuadro verde). **Derecha:** nodos y *topics* ROS que se ejecutan en la Laptop Dell Precision M4500 (recuadro morado).

Este trabajo presenta el diseño e implementación de una simulación Hardware-in-the-loop (HIL). La simulación HIL en este caso implica la interacción entre el módulo *hardware* Jetson TX2 y un simulador de un cuadricóptero UAV para el que no se maneja un *hardware* real. La Figura 2.10 muestra un esquema general de los nodos implementados en ROS y cómo estos se comunican a través de *topics*.

2.8. RViz

ROS *visualization* o RViz es una herramienta de visualización 2D y 3D para ROS. Permite a los usuarios visualizar las simulaciones de modelos de robots, hacer el seguimiento de sistemas de coordenadas 3D (TF, Tansform Data) e información que registran los sensores de un robot. Gracias a esto, el usuario puede depurar la aplicación de acuerdo a las entradas de sensores y acciones planificadas. RViz muestra a través de nubes de puntos o imágenes de profundidad, los datos de sensores como láser (*Laser Imaging Detection and Ranging (LIDAR)*) [89], cámaras estéreo y cámaras de profundidad como por ejemplo, el sensor Kinect [90] o la cámara 3D de profundidad ZED[7] [91]. La interfaz gráfica de usuario que proporciona RViz permite seleccionar fácilmente entre diferentes opciones para mostrar solo la información que el usuario necesite [92]. RViz permite crear y representar modelos de robots, a través de un formato de descripción de robot unificado (*Unified Robot Description Format (URDF)*) [93], [94], [95].

2.9. Gazebo

Gazebo[8] es un entorno de simulación de robots 3D de código abierto desarrollado por Willow Garage, aunque Gazebo usa por defecto el motor de física *Open Dynamics Engine* (ODE)[9],

[7] https://www.stereolabs.com/
[8] http://gazebosim.org/
[9] https://www.ode.org/

también soporta otros motores y bibliotecas como *Bullet*[10], *Dinamic Animation and Robotics Toolkit* (DART)[11], y Simbody[12]. Gazebo proporciona un ambiente que permite la comunicación con ROS, diseños de modelos de robots complejos, prototipado rápido, prueba de algoritmos, simulación en ambientes interiores y exteriores, simulación de datos de sensores y cámaras, y posee una gran cantidad de objetos y escenarios 3D que pueden ser importados.

2.10. OpenCV

OpenCV[13] (*Open Source Computer Vision Library*) es una librería para visión por computador de código abierto, que soporta interfaces de programación como C, C++, Python y Java, y además puede ejecutarse en los sistemas operativos Linux, Windows, Mac OS, iOS y Android. Fue diseñada para obtener alta eficiencia computacional de aplicaciones en tiempo real [96].

La librería cuenta con una gran cantidad de funciones que permite realizar procesos desde los más simples como la umbralización, hasta los más complejos como aplicación de filtros especializados o la creación de sistemas de aprendizaje de máquina. Incluye herramientas para la calibración de cámaras y la creación de interfaces para el usuario.

OpenCV surgió como una iniciativa de investigación de Intel para aplicaciones intensivas de CPU. Los objetivos de esta librería son entre otros: brindar un código abierto y optimizado para infraestructura de visión básica, entregar una infraestructura común que mejore la diseminación del conocimiento de visión por computador, y por último, busca impactar en el desarrollo de aplicaciones comerciales con código portable, optimizado y a bajo costo.

2.11. Python

Es un lenguaje de programación dinámico e interpretado que busca simplificarle al usuario su aprendizaje, para el desarrollo rápido de diversas aplicaciones. Python cuenta con estructura de datos eficientes, de alto nivel y está orientado a objetos. Posee una extensa biblioteca y una gran variedad de módulos y herramientas que están a libre disposición. Está disponible para Linux, Windows y Mac, bajo una licencia de código abierto.

El artículo "*The Top Programming Languages 2020*" publicado por la revista IEEE Spectrum[14], muestra un *ranking* con los lenguajes de programación principales y más usados en la actualidad. Este artículo contiene una clasificación interactiva de acuerdo al campo y tipo de desarrollo en el que se usa el lenguaje de programación. Los *ranking* 2018, 2019 y 2020, muestran a Python como el lenguaje de programación más utilizado en el desarrollo de aplicaciones de inteligencia artificial embebidas. La popularidad de este lenguaje se ha incrementado en la última década, superando a otros lenguajes importantes como Java, C y C++.

[10] https://pybullet.org/wordpress/
[11] https://dartsim.github.io/
[12] https://simtk.org/projects/simbody/
[13] https://opencv.org/
[14] https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020

2.12. TensorFlow

TensorFlow[15] es una plataforma de código abierto para aplicaciones de aprendizaje automático creada por *Google Brain*. Ofrece un ecosistema de herramientas, bibliotecas y recursos para el desarrollo de aplicaciones *machine learning*, en el ámbito de la investigación y de producción. TensorFlow es ampliamente usada como una tecnología para solucionar problemas importantes de aprendizaje automático.

Empresas como Google, Intel, Texas Instruments, Airbnb, Coca-Cola, DeepMind, Twitter, Lenovo, Uber, Paypal, entre otras grandes compañías, usan TensorFlow para para solucionar problemas de su modelo de negocio, o también en la mejora de sus productos y servicios. El flujo de datos gráficos son un elemento central de TensorFlow, donde los nodos del gráfico representan operaciones matemáticas como convolución y reducción (*pooling*) y los bordes del gráfico sirven como matrices de datos multidimensionales (tensores) que se comunican entre ellos. TensorFlow puede mapear tales gráficos sobre diferentes dispositivos como móviles, cluster CPU/GPU. TensoFlow proporciona una API que puede ser accedida a través de Python, JavaScript, C, C++, Java, Go Y Swift, sin embargo, en la documentación oficial, se advierte que las APIs en lenguajes diferentes a Python no están cubiertas por *API stability promises*.[16]

2.13. TensorFlow Object Detection API

La API de detección de objetos TensorFlow es un *framework* que permite la creación de redes *deep learning* que buscan resolver los problemas que se presentan en la detección de objetos [97]. Existen modelos pre-entrenados como el modelo de detección Zoo[17]. La diferencia entre los diferentes modelos de detección de objetos radica en la compensación entre velocidad y precisión que ofrece cada detector [98]. Estos modelos han sido entrenados previamente en el conjunto de datos COCO[18] [99], KITTI[19] [100], Open Images Dataset[20] [101] y AVA[21] [102]. Estos modelos pueden usarse como inferencias cuando se desea usar categorías en específico de los conjuntos de datos. También son útiles, cuando se desea entrenar un modelo en un nuevo conjunto de datos, como es el caso de este proyecto.

2.14. NVIDIA TensorRT

TensorRT es un *Software Development Kit (SDK)* para inferencias de aprendizaje profundo de alto rendimiento en GPU NVIDIA. Ofrece un optimizador de inferencias *deep learning*, bajas latencias en los tiempos de ejecución y alto rendimiento para aplicaciones de inferencia

[15] https://www.tensorflow.org/
[16] https://www.tensorflow.org/api_docs
[17] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md
[18] https://cocodataset.org/
[19] http://www.cvlibs.net/datasets/kitti/
[20] https://storage.googleapis.com/openimages/web/index.html
[21] http://research.google.com/ava/

de aprendizaje profundo. TensorRT permite que las aplicaciones funcionen más rápido que las plataformas con solo CPU durante la inferencia [103].

TensorRT se basa en CUDA, el modelo de programación paralela de NVIDIA, esto permite optimizar inferencias en los *frameworks* de *deep learning* principales y utilizar las librerías, herramientas y tecnologías en CUDA-X para inteligencia artificial y computación de alto rendimiento. Algunos *frameworks* como TensorFlow ya integran soporte de TensorRT con el fin de poder usarse para acelerar la inferencia dentro del *framework*.

Este SDK proporciona optimizaciones con diferentes precisiones como INT8, FP16 y FP32 para el despliegue de aplicaciones de aprendizaje profundo. FP32 y FP16 ofrecen buenos resultados en los que se mejora el rendimiento de la inferencia sin que se vea afectada la precisión de la red. Por su parte, INT8 ofrece una brecha mayor entre la precisión lograda en el entrenamiento de la red y la precisión de la inferencia que se obtiene como resultado. En general, estas inferencias reducen de forma significativa la latencia de las aplicaciones, por lo que son utilizadas en aplicaciones que requieren ejecución en tiempo real [104].

2.15. NVIDIA CUDA

Compute Unified Device Architecture (CUDA) es una plataforma de computación en paralelo y un modelo de programación desarrollado por NVIDIA para la computación en general en unidades de procesamiento gráfico (GPU). CUDA y su conjunto de herramientas, proporciona a los desarrolladores todo lo necesario para acelerar las aplicaciones y aprovechar las propiedades de las GPU. Los desarrolladores que usan CUDA, programan en lenguajes como C, C++, Fortan, Python y Matlab, y expresar el paralelismo a través de la adición de palabras clave necesarias para aprovechar las GPU. CUDA proporciona a los programadores especificar qué parte del código se ejecutará en la CPU y qué parte en la GPU [105].

CUDA ha tenido un gran crecimiento en la última década. La plataforma es utilizada por una amplia variedad de aplicaciones en diversos campos de investigación y sectores como la visión por computador, imágenes médicas, informática financiera, bioinformática, química computacional, meteorología, automatización de diseño electrónico (*Electronic Design Automation (EDA)*), gobierno y defensa, entre otros. En la visión por computador y el procesamiento de imágenes, los algoritmos utilizados son computacionalmente exigentes, dada la gran cantidad de información que se debe procesar. La plataforma permite acelerar estas aplicaciones con el fin de que tengan un rendimiento óptimo. CUDA y una GPU, permiten entrenamientos más rápidos de redes neuronales artificiales y algoritmos de aprendizaje profundo [106].

2.16. NVIDIA cuDNN

cuDNN[22] es una biblioteca GPU-acelerada de primitivas para redes neuronales. Proporciona implementaciones altamente ajustadas para rutinas estándar como convolución hacia adelante

[22] https://developer.nvidia.com/cudnn

y hacia atrás, reducción o *pooling*, normalización y capas de activación. cuDNN acelera los *frameworks* más populares en el *deep learning*, algunos como TensorFlow,[23] Keras,[24] PyThorch,[25] MxNet,[26] Matlab y Caffe2 [107]. Otra característica, incluye la optimización de núcleos para modelos de visión y procesamiento de lenguaje natural (*Natural Language Processing (NLP)*) por computador, como ResNet [108], ResNext [109], SSD [87], MaskRCNN [110], U-Net [111], V-Net [112], BERT [113], GPT-2 [114] y WaveGlow [115]. cuDNN es compatible con Windows y Linux, con arquitecturas GPU como Ampere, Turing, Volta, Pascal, Maxwell y Kepler [116].

2.17. Módulo de desarrollo

Los (*Single Board Computer (SBC)*) y (*System on Module (SOM)*) son ordenadores embebidos completamente funcionales y contenidos en una placa. A diferencia de la placa base de una computadora personal, que combina más de una tarjeta para realizar diferentes tareas, este tipo de ordenadores busca reunir las mismas funcionalidades en una sola placa.

Estos ordenadores han ganado gran aceptación en el mercado por su bajo costo, su uso se ha extendido en la comunidad académica e investigativa, para desarrollar aplicaciones para control de procesos, robótica móvil e industrial, adquisición de datos y proyectos militares. Son ideales cuando el trabajo que se desea realizar tiene tareas específicas, que requieren cierto nivel de demanda computacional en un ordenador que sea pequeño y de poco peso para facilitar su transporte.

Cada uno de estos ordenadores incluye procesador, memoria RAM, variedad de interfaces de comunicación, puertos de red y pines de entrada/salida de propósito general (*General Purpose Input/Output (GPIO)*). Los procesadores van desde los basados en la arquitectura X86 como AMD e Intel, utilizados tradicionalmente por computadores personales, hasta procesadores que basan su arquitectura en ARM, que se encuentran en sectores como el industrial y más recientemente en dispositivos electrónicos como teléfonos móviles, relojes inteligentes, videoconsolas, reproductores de música y discos duros, entre otros. Los procesadores desarrollados por la multinacional ARM Holdings, incluyen las series de la familia ARM y las series Cortex.

El sistema operativo más comúnmente soportado por este tipo de módulos es Linux, en las distribuciones más comunes como Ubuntu, Debian, Fedora, OpenSUSE y Mandriva, entre otras. Las herramientas para la programación y depuración generalmente hacen parte de la iniciativa *open source*, lo que ha permitido una mayor producción y distribución de este tipo de ordenados. Los SBC de código abierto han permitido que las comunidades de desarrolladores fortalezcan sus propuestas, proporcionen actualizaciones de *software* y den a conocer proyectos que mejoran la funcionalidad de estos ordenadores. Algunos de los SBC con mayor aceptación

[23] https://www.tensorflow.org/
[24] https://keras.io/
[25] https://pytorch.org/
[26] https://mxnet.apache.org/

son: Raspberry Pi[27], ODROID,[28] UDOO,[29] BeagleBone,[30] LattePanda,[31] y Radxa Rock.[32]

Los avances en la Inteligencia Artificial (IA), *machine learning*, *deep learning* y el internet de las cosas (*Internet of Things (IoT)*), han conllevado a que los fabricantes de tecnología desarrollen plataformas de *hardware* que integran cada vez más en sus diseños, procesadores y recursos capaces de cumplir con las demandas de la industria. Las redes neuronales profundas (*Deep Neural Networks (DNNs)*) son hoy en día una práctica común en la mayoría de las aplicaciones de inteligencia artificial. El objetivo de este tipo de aplicaciones consiste en tomar decisiones en función de los datos procesados, por ejemplo, en los sistemas de vehículos autónomos, la navegación con drones y la robótica. Para muchas de estas aplicaciones, se prefiere el procesamiento embebido local por encima de la nube, ya que la latencia y los problemas de seguridad y privacidad al depender de la nube representan un mayor riesgo [117]. Esto ha motivado el desarrollo de numerosas técnicas de optimización a nivel de *hardware* y *software*, y arquitecturas especializadas, para procesar modelos *deep learning* con alto rendimiento y eficiencia energética sin afectar su precisión [118].

Las cámaras inteligentes, los sensores de alta resolución, radares, teléfonos inteligentes, drones, vehículos autónomos, robots comerciales, las interfaces controladas por comandos de voz, por visión o por gestos, *chatbots* entre otros, son sistemas que recopilan grandes cantidades de información que requiere ser procesada en tiempo real. Grandes compañías como Google, NVIDIA, Intel, AMD y Qualcomm continúan especializándose en desarrollar e incorporar plataformas integradas para manejar cargas de trabajo de aprendizaje automático que van desde modelos pequeños, medios y de *deep learning*. NVIDIA[33] se destacada en el mercado por sus desarrollos de GPU y otros productos integrados que han acelerado la investigación, el desarrollo y la implementación de soluciones de IA.

Los sistemas de NVIDIA Jetson[34] son una línea de plataformas embebidas que proporcionan el rendimiento y la eficiencia energética para ejecutar *software* de máquinas y robots autónomos, de forma rápida y con menos energía. Cada uno de estos sistemas SOM son completos, tienen CPU, GPU, variedad de insterfaces de comunicación (Ethernet, WiFi, USB, HDMI, SPI, I2C, y UART), circuitos integrados de administración de energía (*Power Managment Integrated Circuits (PMIC)*), memoria dinámica de acceso aleatorio (*Dynamic Random Access Memory (DRAM)*) y almacenamiento flash, características que hacen de los SOM, sistemas ampliamente usados en aplicaciones de inteligencia artificial como la inspección óptica automatizada (*Automated Optical Inspection (AOI)*) y los robots móviles autónomos (*Autonomous Mobile Robots (AMR)*) con necesidades específicas. Dentro del portafolio de Jetson, están los módulos Jetson TK1, Jetson Nano, Jetson TX1, Jetson TX2, Jetson AGX Xavier y Jetson Xavier NX.

[27] https://www.raspberrypi.org/
[28] https://www.hardkernel.com/
[29] https://www.udoo.org/
[30] http://beagleboard.org/bone
[31] http://www.lattepanda.com/
[32] http://wiki.radxa.com/Rock
[33] https://developer.nvidia.com/embedded-computing
[34] https://developer.nvidia.com/embedded/jetson-developer-kits

3 Metodología

Este capítulo detalla los procedimientos, las pruebas y la metodología utilizada para desarrollar una solución óptima ante el planteamiento del problema y los objetivos de este trabajo. Se propone el desarrollo de un sistema de visión por computador embebido que permita detectar y seguir un objetivo móvil desde un cuadricóptero UAV, utilizando ROS e incorporando sus propiedades en la interfaz de comunicación y simulación del sistema.

El desarrollo del proyecto se enmarca en un modelo de prototipo, también llamado desarrollo con prototipación, que consiste en crear un sistema experimental de forma rápida y a un bajo costo [119]. Este modelo inicia con la definición de objetivos generales del *software*, luego se identifican los requerimientos y las áreas donde se necesita más definición. El objetivo principal de un modelo por prototipo es dar al usuario una vista preliminar del *software* o del sistema. Se le denomina prototipo a la versión funcional del sistema o una parte de este, con el que los usuarios interactúan para tener una idea más clara de los requerimientos del sistema. Una vez puesto en funcionamiento, el prototipo se ajusta de forma gradual hasta que cumpla de manera más precisa con los requerimientos de los usuarios (ver Figura 3.1).

El proceso de construir un diseño preliminar, realizar pruebas, refinar y nuevamente probar se denomina proceso iterativo, debido a que los pasos requeridos para crear un sistema se repiten varias veces hasta satisfacer las necesidades de los usuarios. Este modelo incremental promueve los cambios del diseño del sistema y supone una versión cada vez más precisa y cercana a los requerimientos definidos por los usuarios. Este modelo permite una evaluación y un mejoramiento continuo de las versiones de diseño propuestas.

A continuación, se detalla en qué consiste cada una de las etapas en el desarrollo de un modelo por prototipo:

Identificar los requerimientos: El diseñador del sistema reúne las necesidades y requerimientos del sistema.

Diseño rápido de un prototipo: Esta etapa se centra en el diseño rápido de una versión preliminar del sistema.

Construcción de un prototipo funcional: Se toma el modelo diseñado en la anterior eta-

pa y se realiza una construcción rápida del prototipo funcional.

Evaluación del prototipo: En esta etapa se interactúa y trabaja con el sistema construido, para lograr determinar con qué precisión el prototipo cumple con los requerimientos identificados en la primera etapa.

Ajustes del prototipo: Luego de la retroalimentación recibida por el usuario al momento de usar el sistema. Se recopilan todos los cambios y se procede a ajustar el prototipo para mejorar su calidad. Las últimas dos etapas se repiten hasta que el prototipo satisfaga los requerimientos planteados en la etapa inicial.

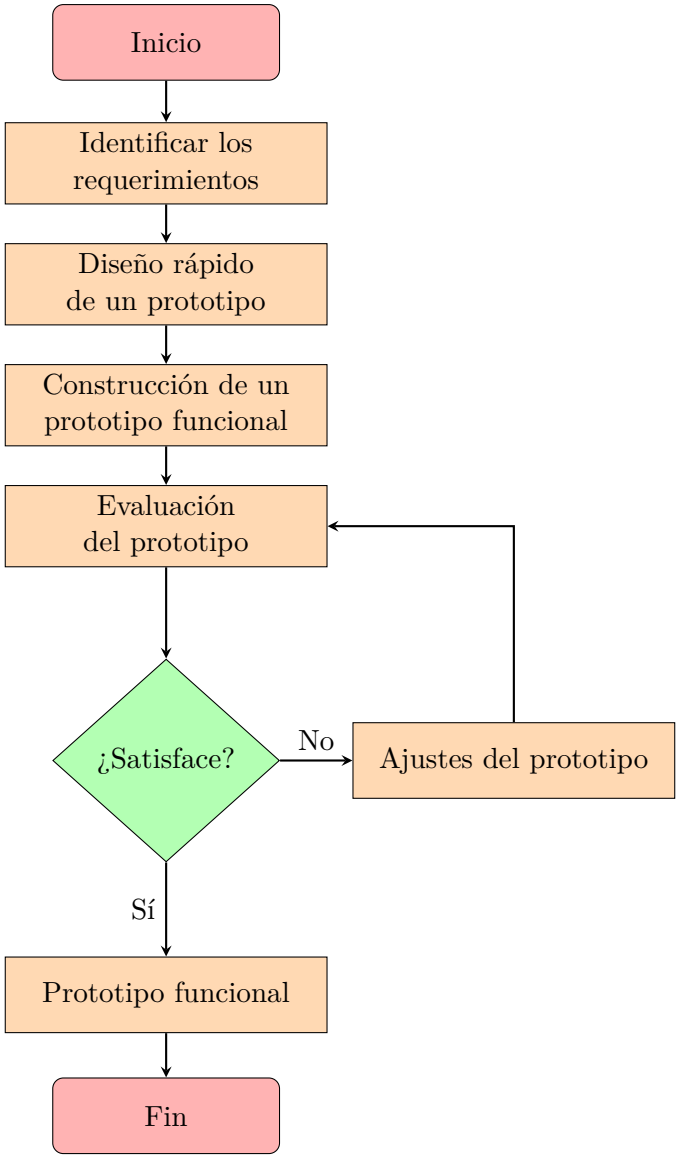


Figura 3.1: Diagrama de flujo del proceso de desarrollo de un prototipo.

Esta metodología es útil cuando no se conocen al detalle los requerimientos o las soluciones de diseño. Al obtener una retroalimentación constante del usuario, existe mayor probabilidad de producir sistemas que cumplan con los requerimientos del usuario. Otra ventaja, consiste en que esta metodología ofrece un mejor enfoque y permite manejar cierto grado de incertidumbre en el funcionamiento de los algoritmos, la adaptabilidad y la forma en que interactúa el usuario con el sistema.

3.1. Análisis e identificación de los requerimientos

Los requerimientos presentados tienen correspondencia con los objetivos propuestos en este trabajo de grado. A continuación se listan los requerimientos del sistema:

- El sistema de visión por computador embebido será capaz de detectar y hacer el seguimiento de un objeto móvil terrestre visto desde un UAV, utilizando ROS.
- Conformar un sistema de visión por computador embebido, de tal modo que pueda ser transportado por un cuadricóptero UAV.
- Configurar el módulo de desarrollo Jetson TX2 con los componentes de *software* necesarios para el funcionamiento del sistema de visión por computador embebido.
- Desarrollar e implementar sobre Jetson TX2, algoritmos y técnicas de procesamiento de imágenes, que permitan la detección y seguimiento de un objetivo móvil terrestre.
- Implementar una interfaz para la comunicación del ordenador embebido con el cuadricóptero UAV, usando ROS.
- Controlar el cuadricóptero UAV para que siga el objeto móvil terrestre, haciendo uso de la información enviada por el sistema de visión por computador.

3.2. Diseño del prototipo

Esta etapa consiste en seleccionar los componentes de *hardware* y *software* necesarios para conformar el sistema embebido. La Figura 3.2 muestra el diagrama de *hardware* del prototipo, su configuración consta de la cámara C922 Pro Stream de Logitech, el módulo Jetson TX2 y la Laptop Dell Precision M4500 utilizada para llevar a cabo la simulación del cuadricóptero.

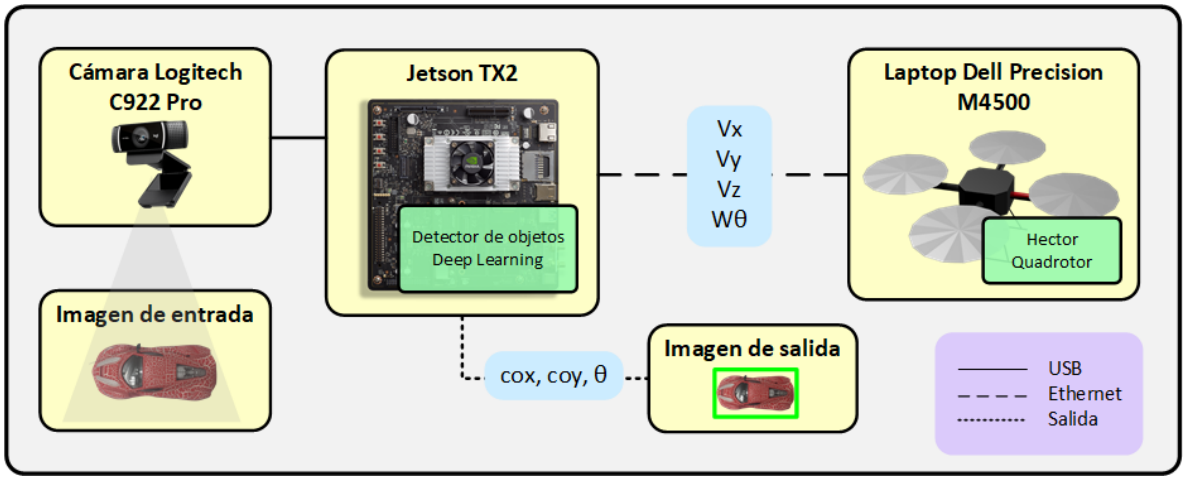
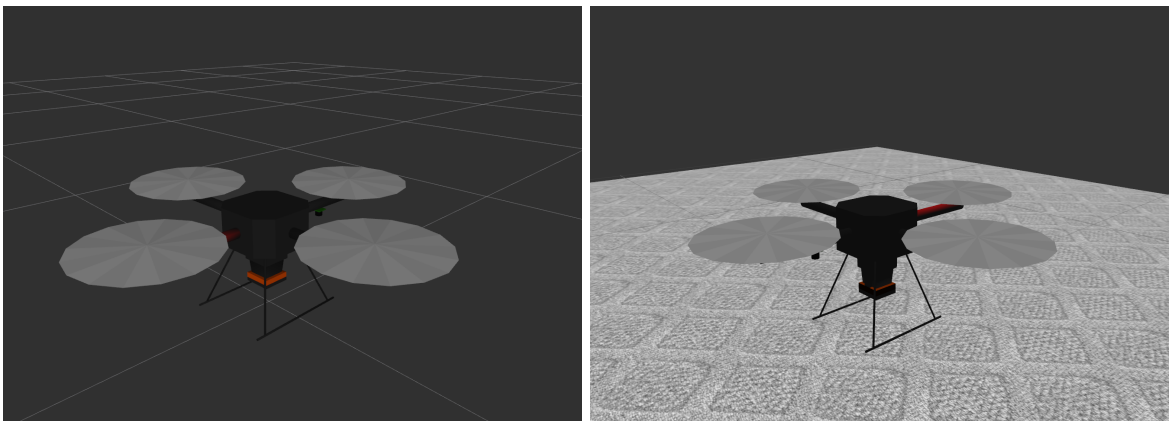


Figura 3.2: Diagrama de *hardware* del sistema.

3.3. Cuadricóptero UAV

Para efectos de implementación y pruebas con una plataforma aérea, se utilizó el metapaquete de ROS *hector quadrotor* [120], que proporciona un modelo en 3D, control y simulación de un cuadricóptero. El modelo está compuesto por cuatro motores brushless DC independientes y hélices de paso fijo, cada par de hélices opuestas giran en una dirección. Incluye varios sensores para estimar la posición, la altitud y la velocidad del UAV. Estos sensores hacen parte del modelo del robot (*URDF*) y permiten ser visualizados en RViz y en Gazebo, como se muestra en la Figura 3.3a y Figura 3.3b, respectivamente. Contiene un nodo que permite el control teleoperado desde un mando de videojuegos o desde el teclado.



(a) (b)

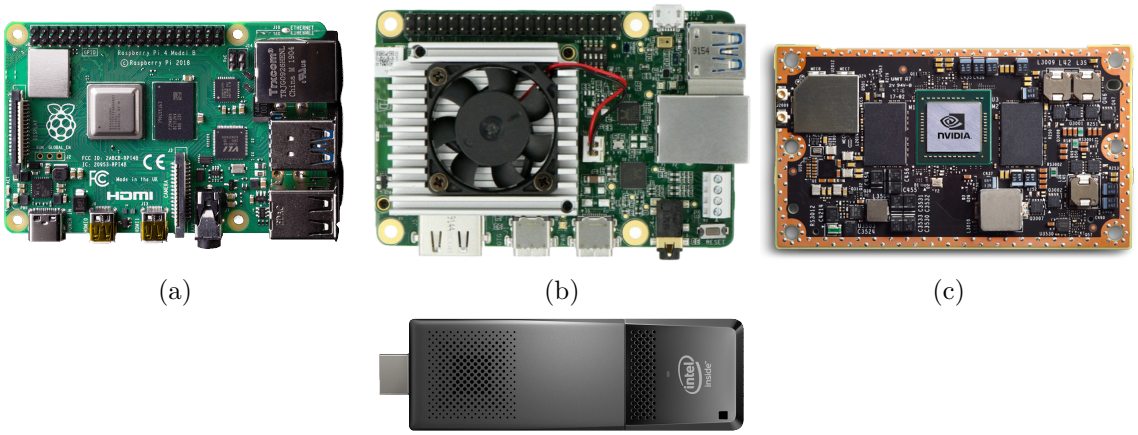
Figura 3.3: Modelo 3D de cuadricóptero *hector quadrotor*: **a:** Modelo en RViz y **b:** Modelo en Gazebo.

Para estimar la pose, la posición y la velocidad del UAV, son necesarios varios sensores

que se han implementado como complementos independientes en Gazebo y se pueden adjuntar al modelo incluyéndolos en la descripción URDF del robot. Algunos de estos sensores son: la unidad de medida inercial (*Inertial Measurement Unit (IMU)*), sensor barométrico, sensor ultrasónico, sensor de campo magnético y receptor GPS. La simulación del cuadricóptero puede llevarse a cabo en diferentes escenarios exteriores e interiores.[35]

3.4. Selección de *hardware*

La elección del módulo de desarrollo consideró las investigaciones realizadas en [121], [118], que proporcionan una descripción general de diferentes arquitecturas *hardware* y técnicas de optimización para la ejecución eficiente de algoritmos de aprendizaje profundo. Adicional, se realizó un comparativo entre algunas de las plataformas *hardware* del mercado para desarrollo *deep learning* (ver Tabla 3.1)[36]. La Figura 3.5 muestra una descripción detallada de las principales características del módulo de desarrollo Jetson TX2, plataforma seleccionada para este proyecto. Este dispositivo informático de inteligencia artificial está basado en una (*Graphics Processing Unit (GPU)*) de la familia NVIDIA Pascal, integra 256 núcleos, una CPU (*Central Processing Unit (CPU)*) ARMv8 de 64 bits de núcleo y una memoria RAM LPDDR4 de 8 GB. La CPU combina un NVIDIA Denver 2 de doble núcleo con un ARM Cortex-A57 de cuatro núcleos[37]. Jetson TX2 es ideal para aplicaciones de aprendizaje profundo, procesamiento de imágenes, drones, robots, realidad aumentada, realidad virtual y dispositivos médicos portables, en su mayoría son aplicaciones que requieren una capacidad de respuesta casi que en tiempo real y con una latencia mínima. NVIDIA que habilita un conjunto de herramientas de desarrollador con (*Application Programming Interface (API)*) como CUDA Toolkit 8.0.56, TensorRT 1.0 GA, VisionWorks 1.6, Tegra System Profiler 3.7 y Tegra Multimedia API.



(a) (b) (c)

(d)

Figura 3.4: Módulos *hardware* embebidos para aplicaciones *deep learning*; **a:** Raspberry Pi 4, **b:** Coral Dev Board, **c:** Jetson TX2, **d:** Intel Compute Stick.

[35] http://wiki.ros.org/hector_quadrotor
[36] https://hackerboards.com/compare/?ids=491,270,29,203,
[37] https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

Propiedad	**Módulo**			
	Raspberry Pi 4	**Coral Dev Board**	**Intel Compute Stick**	**Jetson TX2**
Fabricante	Raspberry Pi Foundation	Google	Intel	NVIDIA
Tipo de tarjeta	SBC	SBC	Stick PC	Módulo
Rendimiento AI	13.5 GFLOPs	32 GFLOPs	100 GFLOPs	1.3 TFLOPs
CPU	Quad-core ARM Cortex-A72	Quad-core ARM Cortex-A53	Quad-core Intel Atom	Quad-core ARM Cortex-A57 NVIDIA Denver2
Velocidad	1.5 GHz	1.5 GHz	1.33 GHz	2 GHz
Núcleos	4	4	4	6
Bits	64	64	32	64
GPU	Broadcom Video Core VI	Vivan GC7000 Lite	Intel HD Graphics	Nvidia Pascal 256 CUDA cores 1300MHz
Memoria	8 GB	1 GB	1 GB	8 GB
Almacenamiento	32 GB	8 GB	8 GB	32 GB
Puertos LAN	1	1	0	1
WiFi	802.11	802.11ac 2x2 MIMO 2.4	802.11b	802.11a 2x2 867Mbps
Pines GPIO	28	40	0	400
Entrada de voltaje (mín.)	5 V	5 V	5 V	5.5 V
Entrada de voltaje (máx.)	5 V	5 V	5 V	19.6 V
Peso	45 g	77 g	60 g	85 g
Website del proyecto	Website	Website	Website	Website

Tabla 3.1: Especificaciones de *hardware* embebido disponible en el mercado para aplicaciones *deep learning*.

NVIDIA Jetson TX2

Característica	Descripción
CPU	ARM Cortex-A57 (quad-core) @ 2GHz + NVIDIA Denver2 (dual-core) @ 2GHz
GPU	256-core Pascal @ 1300MHz
Memoria	8GB 128-bit LPDDR4 @ 1866Mhz | 59.7 GB/s
Almacenamiento	32GB eMMC 5.1
Cámara	12 lanes MIPI CSI-2 | 2.5 Gb/sec per lane | 1400 megapixels/sec ISP
Display	2x HDMI 2.0 / DP 1.2 / eDP 1.2 | 2x MIPI DSI
Wireless	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.1
Ethernet	10/100/1000 BASE-T Ethernet
USB	USB 3.0 + USB 2.0
Entrada	5.5 - 19.6 VDC

Figura 3.5: Especificaciones técnicas del módulo de desarrollo NVIDIA Jetson TX2.

3.5. Dispositivo de captura de video

La selección de la cámara web, tuvo en cuenta las características y rendimiento de dos sensores, el modelo C525 y C922 Pro Stream[38]. Las especificaciones del modelo Logitech C922 Pro Stream se muestran en la Figura 3.6. En comparación con el modelo C525, el modelo C922 cuenta con mayor resolución y cuadros por segundo, ofrece un campo de visión más amplio y presenta menor distorsión en el campo de visión. En el Anexo A se presenta el procedimiento para la calibración de la cámara con el uso de funciones provistas por OpenCV.

Cámara web Logitech C922 Pro Stream

Característica	Descripción
Tipo de conexión	USB 2.0
Microfono	Dual
Grabación	1080p30fps, 720p60fps, 720p30fps
Tipo de lente y sensor	Lente de cristal Full HD
Tipo de foco	Autofoco de 20 pasos
Campo de visión	78° diagonal
Velocidad de fotogramas	720p60fps/1080p30fps
Luz	Corrección automática en condiciones de poca luz

Figura 3.6: Especificaciones técnicas de cámara web Logitech C922 Pro Stream.

[38] https://bit.ly/3vCwqUo

3.6. Ambiente de desarrollo y *software*

Este apartado describe los dispositivos de *hardware* utilizados para la implementación del sistema. En la Figura 3.7 se representa el diagrama de software del sistema, mientras que en la Tabla 3.2 se listan los equipos de *hardware* y el *software* con las versiones utilizadas para su desarrollo.

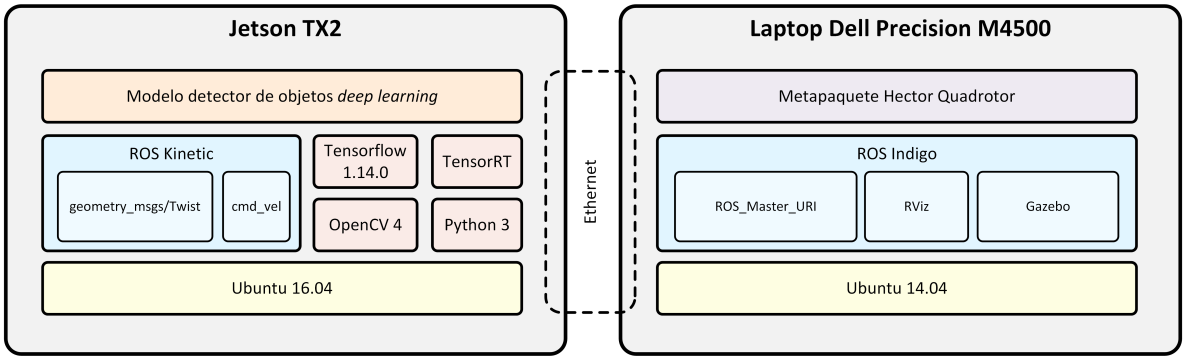


Figura 3.7: Diagrama de *software* del sistema.

Equipo	Software
Laptop Lenovo Legion Y740 *Entrenamiento de modelos* GPU NVIDIA Geforce RTX 2060 CPU Intel Core i7-9750H Memoria: 16 GB DDR4 SDRAM 2666 MHz Sistema Operativo: Windows 10 Home Single	CUDA 10 cuDNN 10 Python 3.7.6 OpenCV 4.0 TensorFlow 1.14.0 TensorRT 4.0 TensorFlow Object Detection API
NVIDIA Jetson TX2 *System-on-Modeule* Sistema Operativo: Ubuntu 16.04 LTS	Jetpack 3.3 CUDA 9.0.252 cuDNN 7.1.5 TensorRT 4.0 Python 3.5 OpenCV 4.0 TensorFlow 1.14.0 TensorFlow Object Detection API ROS Kinetic
Laptop Dell Precision M4500 *Ambiente de simulación* Intel Core i7-M620 Memoria: 8 GB DDR3 Sistema Operativo: Ubuntu 14.04.6 LTS	Python 3.5 RViz ROS Indigo Gazebo Hector Quadrotor

Tabla 3.2: Configuración de ambiente de desarrollo y herramientas de *software* utilizado.

3.7. Evaluación y ajustes del prototipo

Esta fase consiste en la implementación de experimentos que permitan la validación de dos tareas principales: la detección y el seguimiento del objeto móvil terrestre por el cuadricóptero. Además, se busca evaluar el rendimiento y precisión de algoritmos implementados. Antes de llevar a cabo pruebas de operación del sistema en conjunto, se realizaron diferentes experimentos en cada módulo que conforma el sistema. Primero se ejecuta el nodo *master* desde la Laptop Dell Precision M4500 que ejecuta el modelo 3D *hector quadrotor* en RViz y Gazebo con el fin de simular el comportamiento de la plataforma aérea. En la Jetson TX2 se ejecuta el sistema de visión por computador embebido desarrollado y se de un blanco móvil para verificar el funcionamiento, tomar las respectivas mediciones y determinar los posibles fallos del sistema. Con base en los resultados obtenidos durante la evaluación del sistema, en esta etapa también se realizan ajustes y modificaciones al sistema. Posteriormente, se validan nuevamente los resultados obtenidos y se realizan recomendaciones para trabajos futuros en el área.

4 Detección de objetos

En este capítulo se describe la metodología utilizada para el entrenamiento de tres modelos de detección de objetos de aprendizaje profundo (*deep learning*). Se muestran las diferentes etapas desde la conformación del *dataset* personalizado, la anotación de las imágenes, el pre-procesamiento del banco de imágenes, el entrenamiento y la evaluación de los modelos entrenados.

4.1. Banco de imágenes (*dataset*)

El sistema de visión por computador propuesto en este trabajo de grado contempla el desarrollo e implementación de un detector de objetos personalizado, es decir, de un objeto en concreto. La Figura 4.1 muestra el objetivo móvil que se detectará, es un carro de radio control (RC) con dimensiones: 10 [cm] de ancho, 23 [cm] de largo y 6.5 [cm] de alto. El proceso de entrenamiento y aprendizaje de un modelo detector de objetos personalizado requiere la conformación de un banco de imágenes *dataset* numeroso, que comprenda una gran variedad de fotografías del objeto que se desea detectar.

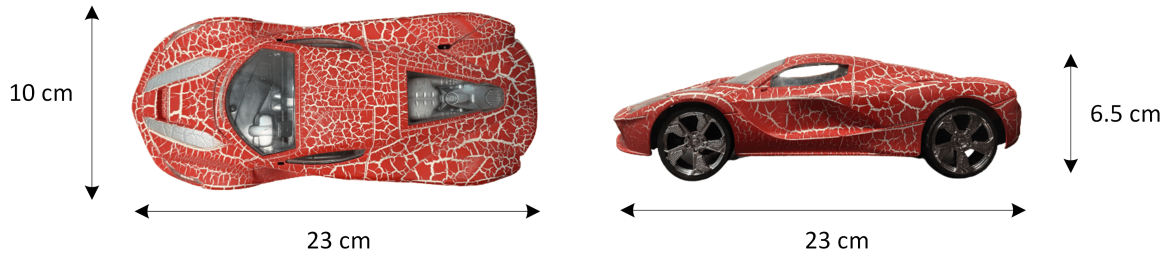


Figura 4.1: Vista superior y lateral del objetivo móvil terrestre, y sus dimensiones. Carro de radio control con dimensiones: 10 [cm] de ancho, 23 [cm] de largo y 6.5 [cm] de alto.

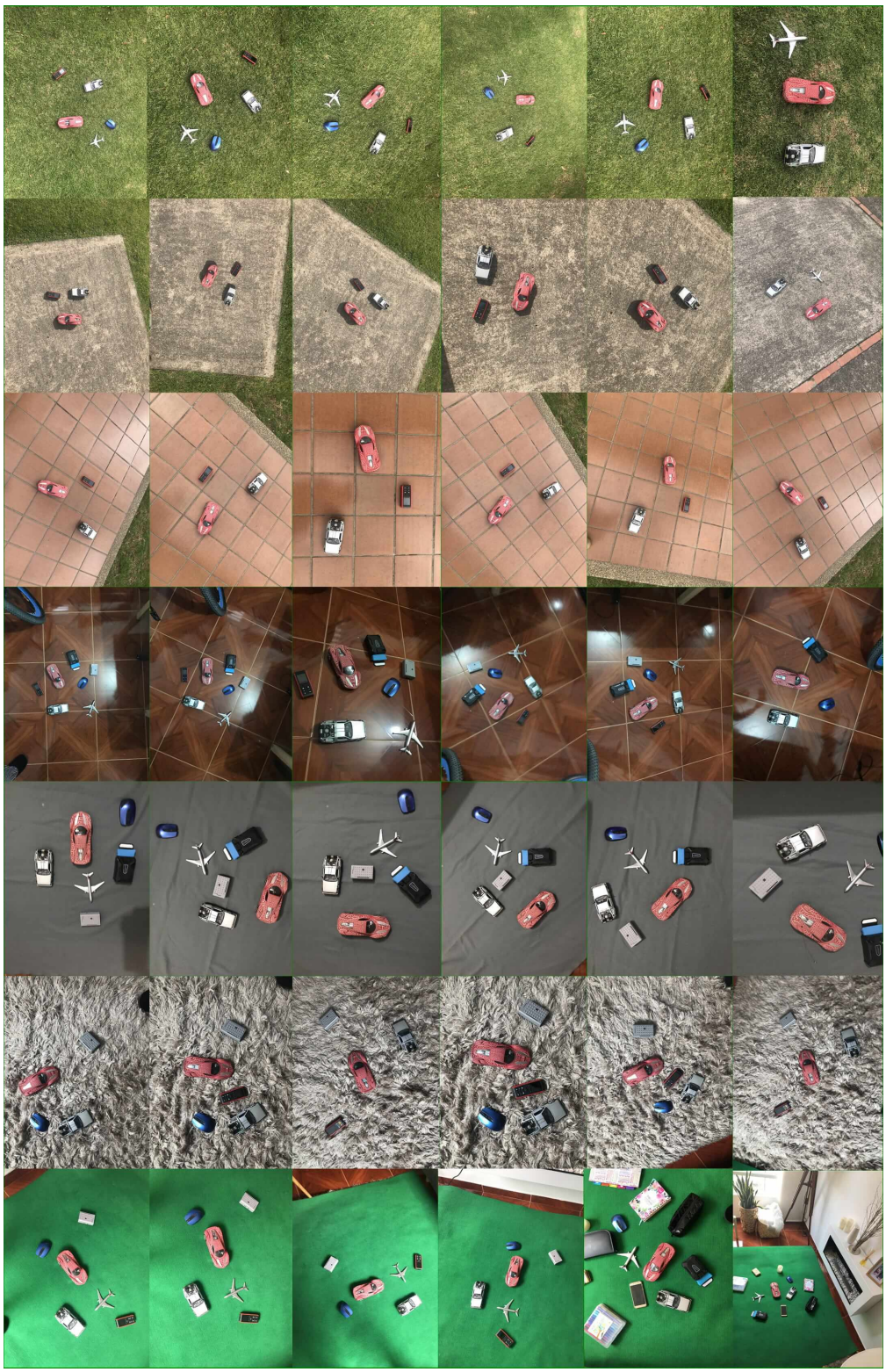


Figura 4.2: Subconjunto de imágenes del carro de radio control tomado del *dataset*. Las fotografías capturadas se tomaron en ambientes exteriores e interiores, con diferentes variaciones en escala, pose, fondos e iluminación.

Para conformar el *dataset*, se recopiló un total de 2396 imágenes del carro de radio control. Las fotografías recopiladas se capturaron con la cámara que incorpora el teléfono móvil iPhone 7 Plus, de acuerdo con sus especificaciones, esta cámara ofrece imágenes fotográficas con una resolución de 12 $[Mpx]$ y un tamaño de imagen de 3024×4032 pixeles. Las fotografías se tomaron en ambientes exteriores como parques, zonas verdes, calles y estacionamientos al aire libre con iluminación natural. Los ambientes interiores corresponden a los espacios de una vivienda como habitación de estudio, sala y pasillos. Un aspecto importante que se debe considerar al crear un *dataset*, es que las fotografías del objeto a detectar tengan grandes variaciones en escala, pose, fondo e iluminación. La Figura 4.2 es un subconjunto de datos con 42 fotografías del carro de radio control, en ambientes exteriores e interiores, con diferentes escalas, poses, fondos e iluminación. Así mismo, para la conformación de escenarios visualmente más complejos, se utilizaron objetos con formas y tamaños similares.

Con el banco de imágenes (*dataset*) ya conformado, se renombran los archivos JPG con una misma plantilla y secuencia establecida. Para logar un procesamiento fluido de las imágenes, se redimensionan los archivos a un tamaño medio aproximado de 540×720 pixeles. Las tareas de renombrar y redimensionar las imágenes de forma masiva se realizaron con la herramienta libre para uso personal y educativo FastStone Image Viewer.[39] Esta práctica de redimensionar las imágenes es recomendable para evitar que se ejecuten futuros errores por falta de memoria durante entrenamiento del modelo.

4.2. Anotación de imágenes

El etiquetado de imágenes es un proceso fundamental en tareas de aprendizaje supervisado, debido a que la calidad de los datos de entrada va a determinar la calidad de los modelos *deep learning* entrenados. Recopilar una amplia gama de imágenes anotadas con el tipo de objeto correcto, va a ayudar a que el modelo aprenda a identificar el objeto deseado. La anotación de imágenes para la detección de objetos, consiste en dibujar un cuadro delimitador sobre el objeto que se desea detectar y asignar un nombre de etiqueta a la clase u objeto.

Con el banco de imágenes conformado, se procede a la anotación de las 2396 imágenes recopiladas. Esta tarea se realizó con la ayuda del *software* libre LableImg[40]. Esta herramienta permite la anotación de imágenes gráficas, dibujar cuadros delimitadores de objetos y asignarle un nombre de etiqueta correspondiente. LabelImg se encuentra escrito en Python y utiliza QT para la interfaz gráfica de usuario, es posible instalarla y ejecutarla en los sistemas operativos Linux, Windows y Mac. Las anotaciones se guardan como archivos *Extensible Markup Language (XML)* en formatos Pascal VOC *Visual Object Classes (VOC)* o *YOLO* (ver Anexo C).

[39] https://www.faststone.org/
[40] https://github.com/tzutalin/labelImg

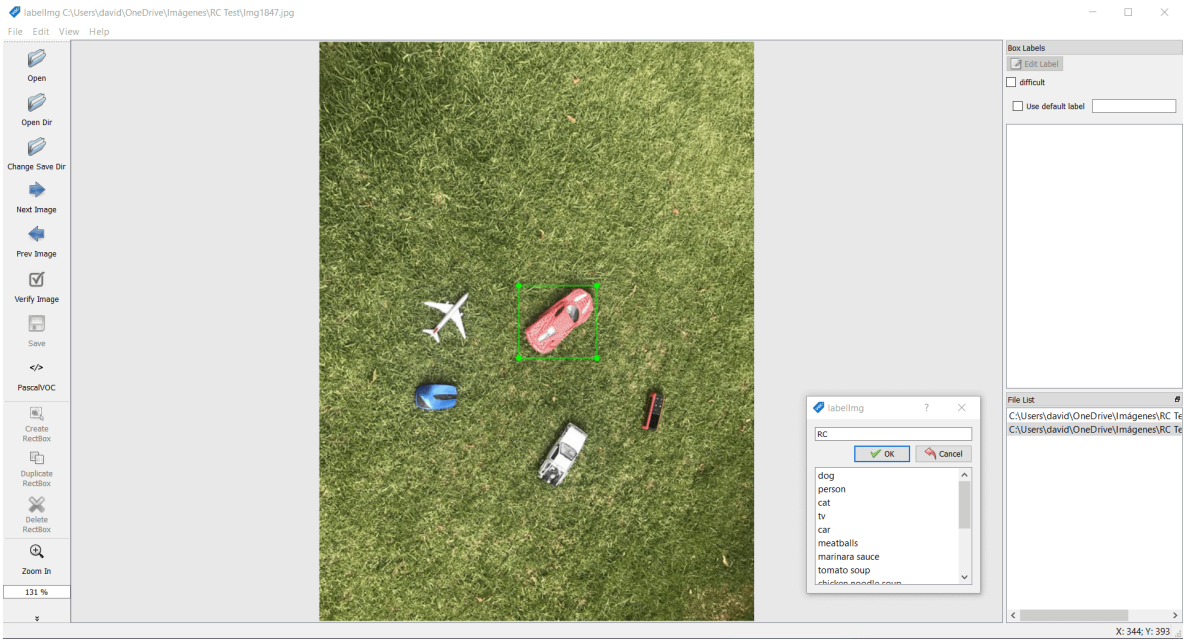


Figura 4.3: Anotación de una imagen del carro de radio control (RC) del *dataset*, con la herramienta LabelImg.

4.3. Pre-procesamiento de datos

Después de completar el proceso de las anotaciones del *dataset* se divide el conjunto de datos en dos: una carpeta que contiene las imágenes para el entrenamiento (*train*) del modelo y otro de prueba (*test*) que contiene las imágenes que se utilizarán para la evaluación del modelo, cada directorio está compuesto por 1910 y 486 imágenes respectivamente, que conforman un *dataset* con 2396 imágenes para una clase que en este caso se denomina como `'RC'`. La distribución se realizó tomando el 80 % de imágenes para el entrenamiento del modelo y el 20 % restante para su evaluación. Esta división se realiza para analizar y hacer un seguimiento de forma más detallada al comportamiento del modelo mientras se está entrenando.

Los modelos *deep learning* trabajan con grandes volúmenes de datos, por lo que el uso de un formato binario para almacenar los datos representa una lectura eficiente que puede tener un impacto significativo en el rendimiento del canal de importación, y como consecuencia en el tiempo de entrenamiento del modelo.

TensorFlow provee varios modelos de detección de objetos que se han entrenado previamente en múltiples conjuntos de datos (*dataset*) de visión por computador más comunes, como COCO,[41] KITTI,[42] Open Images[43] y AVA v2.1.[44]

Se puede elegir entre diferentes modelos pre-entrenados del modelo de detección Zoo de

[41] http://cocodataset.org/
[42] http://www.cvlibs.net/datasets/kitti/
[43] https://storage.googleapis.com/openimages/web/index.html
[44] https://research.google.com/ava/

TensorFlow desde su repositorio[45]. Este proyecto utilizó modelos pre-entrenados en el *dataset* COCO [99], este es un conjunto de datos para la detección, segmentación y subtitulación de objetos a gran escala. COCO está compuesto por más de 200000 imágenes etiquetadas y 80 clases o categorías de objetos.

4.4. Entrenamiento de modelos

Es en esta etapa donde se aplican técnicas de *deep learning* a los modelos de redes neuronales convolucionales (CNN). *Fine-tunning* es un procedimiento basado en el concepto de aprendizaje de transferencia (*transfer learning*). Este tipo de aprendizaje consiste en tomar una red pre-entrenada en un conjunto de datos determinado, en lugar de entrenar una red neuronal desde cero, que implica definir una gran cantidad de datos, además de representar un gran costo computacional de días o semanas para entrenar el modelo.

El ajuste fino o *fine-tunning* permite reemplazar la capa de salida, originalmente creada para reconocer 80 clases en el caso del *dataset* COCO, por una capa que reconoce la cantidad de clases requeridas, que para este proyecto corresponde a una clase. Se eligieron tres modelos pre-entrenados TensorFlow de código abierto para ser presentados en este proyecto: SSD con Inception v2 (`ssd_inception_v2_coco`), Faster R-CNN con Inception v2 (`faster_rcnn_inception_v2_coco`) y SSD con MobileNet v1 (`ssd_mobilenet_v1_coco`).

Con la ayuda de TensorBoard,[46] que proporciona la visualización y las herramientas para trazar gráficos computacionales, se muestra la disminución de pérdida total de cada modelo durante todo el proceso de *fine-tuned*. La pérdida total representa la pérdida de clasificación y localización. La pérdida de clasificación y localización, representan que tan bueno es el modelo clasificando y localizando un objeto de forma correcta. Los valores de pérdida se van consolidando después de un número de pasos determinado.

4.4.1. Entrenamiento del modelo SSD Inception v2

El modelo SSD Inception v2 se entrenó con la técnica *fine-tuned* con un total de 65104 pasos. La Figura 4.4 muestra el valor de pérdida de clasificación y localización disminuye de forma considerable, y por lo tanto, también los valores de pérdida total. Esto se debe gracias a que se eligen modelos pre-entrenados, en lugar de entrenar desde cero. Los valores presentados en color naranja descolorido son los valores reales de pérdida, mientras que el color naranja oscuro representa los valores con un suavizado *smooting* = 0.6.

[45] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md

[46] https://www.tensorflow.org/tensorboard

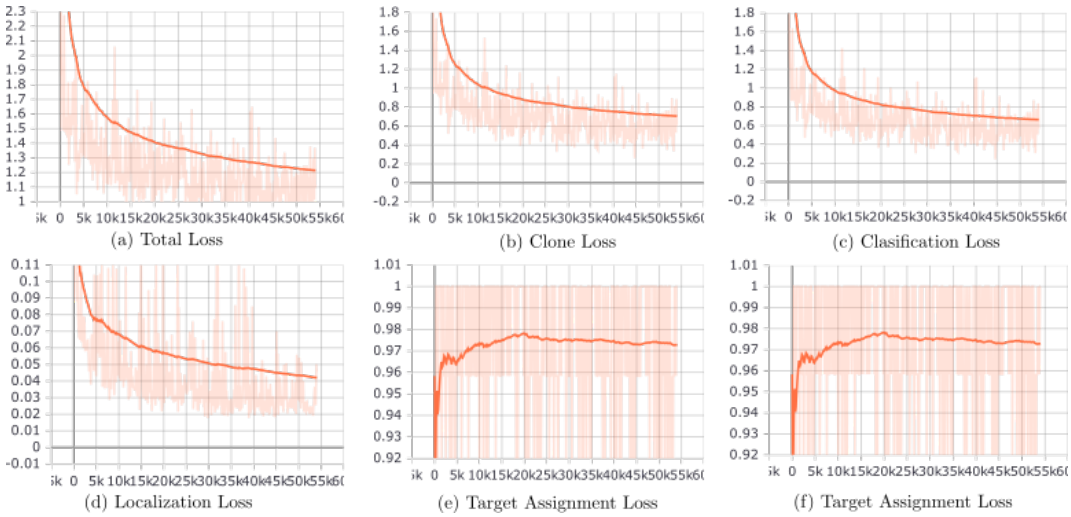


Figura 4.4: Valores en Tensorboard de SSD Inception v2 durante *fine-tuned* con un total de 65104 pasos.

4.4.2. Entrenamiento del modelo SSD MobileNet v1

Otro modelo presentado es SSD MobileNet v1 (`ssd_mobilenet_v1`). Al igual que el anterior, este modelo pre-entrenado también trae un archivo base de configuración para la puesta a punto *fine-tunning*. El modelo SSD MobileNet v1 se ajustó con un total de 50856 pasos como se observa en la Figura 4.5. La Figura 4.5 muestra la pérdida total del modelo, así mismo, se observa como los valores de pérdida de clasificación y localización disminuyen rápidamente, con una tendencia decreciente con pequeñas fluctuaciones menores.

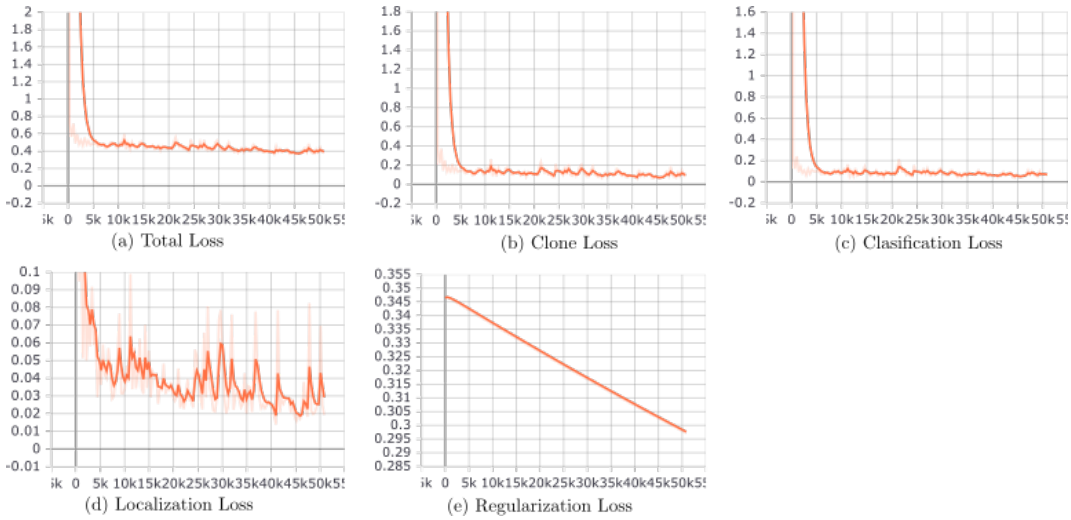


Figura 4.5: Valores en Tensorboard de SSD MobileNet v1.

4.4.3. Entrenamiento del modelo Faster R-CNN Inception v2

El tercer modelo es Faster R-CNN Inception v2 (`faster_rcnn_inception_v2_coco`), al que se le aplica ajuste fino (*fine-tuned*) con el entrenamiento y la validación del *dataset*. La Figura 4.6 muestra el ajuste del modelo Faster R-CNN Inception v2 con un total de 151064 pasos. En este rango, los valores de pérdida total, de clasificación y localización describen una tendencia a la baja con algunas fluctuaciones. En comparación con SSD Inception v2 y SSD MobileNet v1, el modelo Faster R-CNN Inception v2, presenta los valores de pérdida total, clasificación y localización más bajos y cercanos a cero.

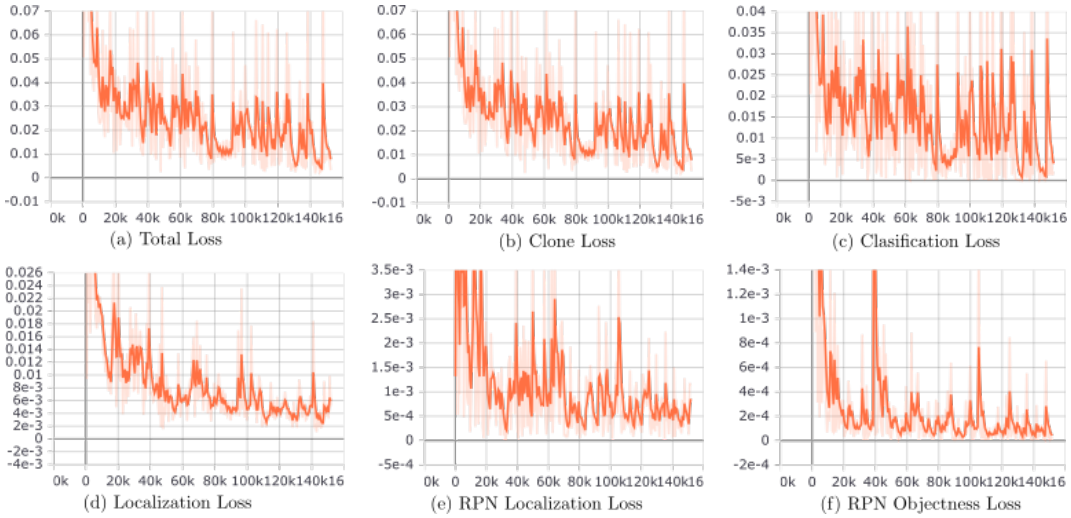


Figura 4.6: Valores en Tensorboard de Faster RCNN Inception v2.

4.5. Métricas de evaluación

La evaluación de un detector de objetos depende de qué tan bien realiza las siguientes tareas: precisión del cuadro delimitador *bounding box*, encontrar los objetos, la precisión en la que detecta el objeto y la salida de la clase correcta para cada objeto presente en la imagen. Existen dos métricas principales que permiten evaluar el rendimiento de un detector de objetos. FPS (fotogramas por segundo) es considerada la métrica más común para medir la velocidad de detección del modelo. Otra métrica común usada para evaluar la tarea de reconocimiento de objetos es mAP (*mean Average Precision (mAP)*), esta métrica genera un porcentaje de 0 a 100 [74]. TensorFlow Object Detection API usa mAP como un protocolo para medir y comparar la precisión de los modelos de detección de objetos.

Más adelante, en el Capítulo 7, se muestran y analizan los resultados de evaluación obtenidos en las métricas FPS y mAP para los tres modelos de detección de objeto seleccionados en este trabajo de grado.

5 Implementación

Las CNN profundas, contienen millones de parámetros que se aprenden durante el entrenamiento del modelo. Los avances a través de la red y el entrenamiento con retropropagación requieren cálculos en todos estos parámetros y gradientes neuronales. Aprovechando las propiedades matemáticas de las convoluciones, se han desarrollado implementaciones en GPU, paralelas y eficientes, como por ejemplo, la biblioteca NVIDIA CUDA Deep Neural Network (cuDNN).

5.1. Optimización de inferencia TensorFlow con TensorRT

La implementación de modelos *deep learning* sobre un computador embebido es uno de los requerimientos principales del sistema. Este tipo de implementaciones representa un reto, si se tiene en cuenta que los recursos de *hardware*, GPU y rendimiento en el procesamiento, son significativamente menores en comparación con los que se pueden alcanzar con computadores de escritorio o portátiles. Con esta finalidad, se optimizaron las inferencias resultantes de los modelos entrenados, mediante TensorRT de NVIDIA. TensorRT es el optimizador de inferencia de aprendizaje profundo que proporciona soporte de precisión mixta, diseño de tensor óptimo, fusión de capas de red y especializaciones del kernel. El módulo de desarrollo Jetson TX2 viene equipado con GPU Volta y soporta precisiones de FP32 o FP16 [122]. Cada modelo se convirtió a TensorRT bajo el modo de precisión FP16 que maximiza los FPS y conserva la precisión (ver Anexo D) [104].

5.2. Máximo rendimiento en Jetson TX2

Jetson TX2, trae consigo la herramienta de línea de comandos `nvpmodel`. Su función es proporcionar diferentes configuraciones de CPU y GPU para ajustar el rendimiento y el consumo de energía. En la Tabla 5.1, se describe cada perfil, los núcleos de CPU que utiliza y la frecuencia máxima de CPU y de GPU que utiliza cada modo.

Propiedad	Modo				
	MAX-N	**MAX-Q**	**MAX-P**	**MAX-P**[1]	**MAX-P**
Modo	0	1	2	3	4
Presupuesto de energía	N/A	7.5W	15W	15W	15W
Frecuencia CPU A57 (MHz)	2000	1200	1400	2000	345
Frecuencia CPU D15 (MHz)	2000	N/A	1400	N/A	2000
Frecuencia GPU (MHz)	1300	850	1122	1122	1122
Frecuencia memoria (MHz)	1866	1331	1600	1600	1600

[1] El modo predeterminado es MAX-P (modo 3).

Tabla 5.1: Modos y configuración de NVPModel para Jetson TX2. Tomado de [123].

En comparación con el modo predeterminado MAX-P (modo 3), MAX-N (modo 0) es la configuración que ofrece el mejor rendimiento en el Jetson TX2 (ver Figura 5.1). En este modo, las frecuencias de reloj de la CPU y la GPU son las más altas. A este perfil, se accede a través del comando: `sudo nvpmodel -m 0`. Los modelos de detección de objetos propuestos, se ejecutaron tanto en el modo predeterminado MAX-P, como en el modo MAX-N.

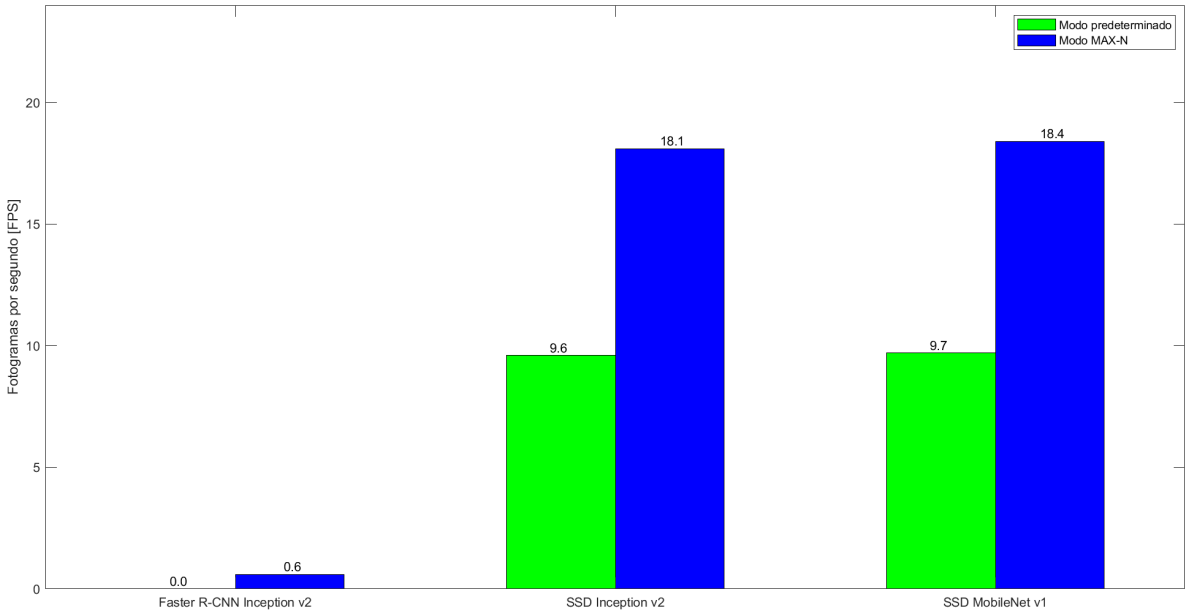


Figura 5.1: Rendimiento de modelos *deep learning* en los modos MAX-P (predeterminado) y MAX-N.

Para la medición de potencia en la Jetson TX2, NVIDIA ha incluido sensores en la placa y en el módulo para realizar este tipo de mediciones de una forma más detallada y precisa. Con la ayuda del programa por consola provisto en [124], fue posible acceder a los sensores y realizar mediciones de potencia, voltaje y corriente. Estas mediciones se tomaron durante la ejecución de uno de los modelos *deep learning*, en el modo MAX-P (predeterminado) y en MAX-N, los resultados se muestran en la Tabla 5.2 y en la Tabla 5.3, respectivamente.

Descripción	**Modo MAX-P**	
	Voltaje [mV]	**Corriente [mA]**
module/main	19184	284
module/cpu	19184	64
module/ddr	4800	292
module/gpu	19192	28
module/soc	19200	44
module/wifi	4800	84
board/main	19184	158
board/5v0-io-sys	5000	536
board/3v3-sys	3336	12
board/3v3-io-sleep	3336	0
board/1v8-io	1816	0
board/3v3-m.2	3336	0
Total		1502

Tabla 5.2: Mediciones de potencia en Jetson TX2 durante la ejecución del modelo *deep learning* en el modo MAX-P (predeterminado).

En la Tabla 5.2, se muestran los resultados de las mediciones de potencia, voltaje y corriente que consume el modo MAX-P en el Jetson TX2 durante la ejecución del modelo *deep learning*. De acuerdo con esto, en el modo MAX-P el valor total de corriente es 1.5 $[A]$ y el de voltaje de 19.2 $[V]$.

Descripción	**Modo MAX-N**	
	Voltaje [mV]	**Corriente [mA]**
module/main	19176	386
module/cpu	19168	88
module/ddr	4800	356
module/gpu	19184	76
module/soc	19184	52
module/wifi	4800	80
board/main	19176	174
board/5v0-io-sys	5000	584
board/3v3-sys	3336	12
board/3v3-io-sleep	3336	0
board/1v8-io	1816	0
board/3v3-m.2	3336	0
Total		1808

Tabla 5.3: Mediciones de potencia en Jetson TX2 durante la ejecución del modelo *deep learning* en el modo MAX-N.

Los resultados de las mediciones de potencia, voltaje y corriente del modo MAX-N se muestran en la Tabla 5.3. En comparación con el modo predeterminado MAX-P, en el modo MAX-N el valor de voltaje aproximado se mantiene en 19.2 $[V]$, mientras que el valor de corriente es de 1.8 $[A]$, es decir, se representa un aumento de 0.3 $[A]$. Estos valores, son útiles para trabajos futuros, que busquen diseñar la batería para suministrar la energía que requiere el sistema transportable por el cuadricóptero y así mismo, para calcular el tiempo de trabajo o de autonomía del sistema.

5.3. Capacidad y uso computacional de Jetson TX2

En esta sección, se analiza el uso computacional requerido por el módulo de desarrollo Jetson TX2 durante la ejecución de las inferencias de los modelos *deep learning* en el modo MAX-N donde se aprovecha el máximo rendimiento.

Figura 5.2: Uso computacional en Jetson TX2; **a:** estado de Jetson TX2 durante la ejecución de la solución *deep learning*, **b:** gráfico de CPU, **c:** historial de GPU, **d:** gráfico de memoria RAM.

Con la ayuda del paquete Jetson Stats[47] y la utilidad Jtop, se monitorea el uso computacional de Jetson TX2. La Figura 5.2 muestra en tiempo real el estado de Jetson TX2 durante la ejecución del sistema de visión propuesto. En la CPU se evidencia que los núcleos están usando entre el $34\,\% \sim 67\,\%$ de su capacidad, el uso de GPU se muestra oscilante con valores entre $20\,\% \sim 90\,\%$, la RAM permanece constante en $2.9\,[GB]/8\,[GB]$, NVPModel en modo MAX-N activo y una potencia promedio de $5.15\,[W]$.

5.4. Posición del objeto

Un detector de objetos retorna en la imagen de salida, una o varias cajas delimitadoras o *bounding boxes*. Cada caja delimitadora es un rectángulo que contiene el objeto que se desea detectar. En los algoritmos detectores de objetos implementados en este proyecto, se obtiene como salida un listado de *bounding boxes* con una única clase denominada `'RC'`, la cual se asigna a la *bounding box* cuando el objeto es detectado. El centroide del carro de radio control (c_{ox}, c_{oy}), se calcula con las coordenadas de salida de cada *bounding box* (x_{min}, y_{min}, x_{max}, y_{max}) que contiene el objeto, como se muestra en la Figura 5.3.

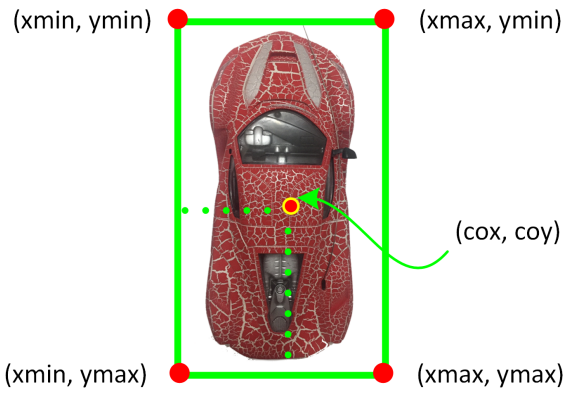


Figura 5.3: Representación de coordenadas de centroide del objeto móvil terrestre.

5.5. Estimación de distancia

Cuando el carro de radio control es detectado, se calculan las coordenadas 2D de su centroide y se estima la distancia euclideana (x) que existe entre la proyección de coordenada centro de la cámara (c_x, c_y) y el centroide del objeto (c_{ox}, c_{oy}) (ver Figura 5.4) con la Ecuación 5.1. Donde c_x y c_y tienen valores de $320\,[pixeles]$ y $240\,[pixeles]$, respectivamente, si se considera una resolución de imagen de entrada de $640 \times 480\,[pixeles]$. Como resultado se obtiene la distancia error en pixeles, para convertir este valor a metros, se incluye un patrón de medida conocida como variable de entrada en el algoritmo de detección, en este caso, se toma el largo del objeto.

[47] https://rbonghi.github.io/jetson_stats/

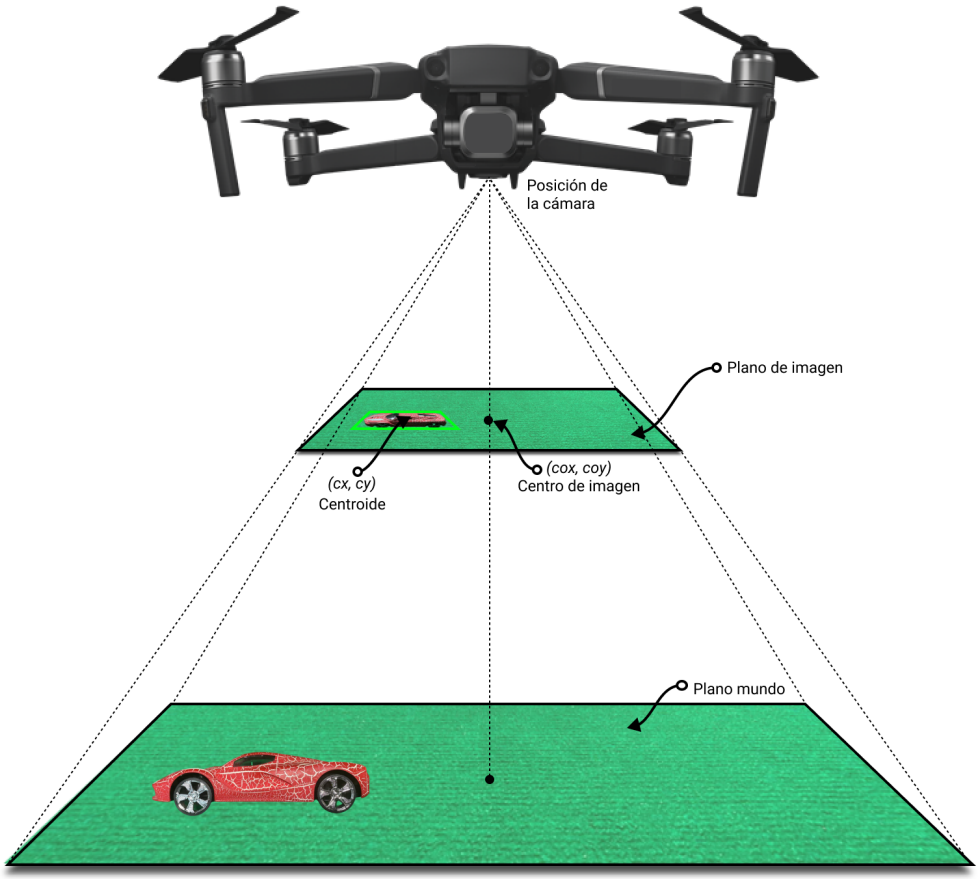


Figura 5.4: Campo de visión desde una cámara a bordo de un cuadricóptero.

$$x^2 = (c_x - c_y)^2 + (c_{ox} - c_{oy})^2$$
$$x = \sqrt{(c_x - c_y)^2 + (c_{ox} - c_{oy})^2} \tag{5.1}$$

$$x\,[m] = l_{object}\,[m]\left(\frac{x\,[px]}{l_{object}\,[px]}\right) \tag{5.2}$$

Luego, se calcula el ángulo que se forma desde el centro del campo de visión de la cámara con respecto al centroide del carro de radio control, como se observa en la Figura 5.5. El ángulo de inclinación de la recta que se forma entre estos dos puntos se calcula con la Ecuación 5.4.

$$tan(\theta) = m = \frac{y_2 - y_1}{x_2 - x_1} \tag{5.3}$$

$$\theta = tan^{-1}(m) \tag{5.4}$$

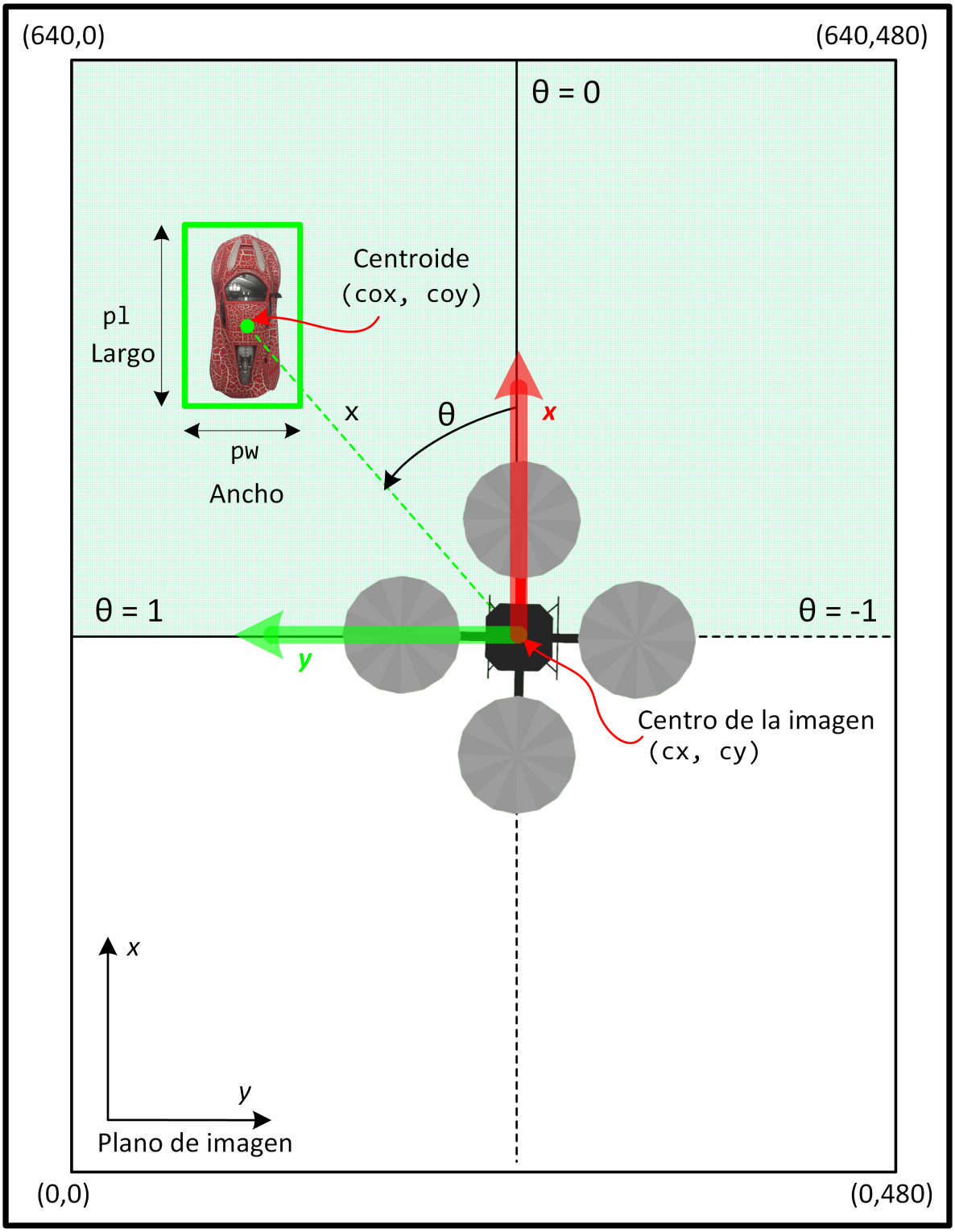


Figura 5.5: Visualización de la proyección de coordenadas desde una cámara a bordo de un cuadricóptero.

5.6. Velocidad Lineal

La velocidad inicial (V_i) se calcula con el valor de la distancia hallado en la Ecuación 5.2 y el tiempo transcurrido entre cada fotograma (t). Para calcular la velocidad lineal que tendrá el cuadricóptero, se calcula la velocidad máxima (V_{max}) que puede alcanzar el carro de radio control cuando se encuentra en el límite superior del campo de visión del fotograma. Con los valores de aceleración (a_f) y la aceleración máxima (a_q) del cuadricóptero, se halla el valor de velocidad lineal total con la Ecuación 5.6. Una vez se halla el vector velocidad lineal V_f del objeto en movimiento, se separa en sus componentes de velocidad lineal en el eje x (V_x) y en el eje y (V_y), con la Ecuación 5.7 y 5.8 respectivamente. La velocidad en el eje z ($V_z = 0$), ya que luego de despegar el robot aéreo no se contempla un control de altitud del cuadricóptero.

$$V_i = \frac{x_{error}}{t} \left[\frac{m}{s}\right] \tag{5.5}$$

$$V_f = \frac{(V_i + (a_f \cdot a_q \cdot t))}{V_{max}} \left[\frac{m}{s}\right] \tag{5.6}$$

$$V_{linearx} = V_f \left[\frac{m}{s}\right] \cdot cos(\theta) \tag{5.7}$$

$$V_{lineary} = V_f \left[\frac{m}{s}\right] \cdot sin(\theta) \tag{5.8}$$

$$V_{linearz} = 0 \left[\frac{m}{s}\right] \tag{5.9}$$

5.7. Velocidad angular

La velocidad angular $\omega_{angularz}$ se calcula utilizando la posición del centroide en pixeles (co_x, co_y) del carro de radio control detectado en el fotograma. El control de operación se divide en dos regiones con ángulos normalizadas en el intervalo [$\theta = -1$, $\theta = 0$] para el primer cuadrante, y en el intervalo [$\theta = 0$, $\theta = 1$] para el segundo cuadrante. De acuerdo con el marco de referencia de la cámara, mostrado en la Figura 5.5, la velocidad angular se considera negativa cuando el carro de radio control está en el primer cuadrante o lado derecho del fotograma, mientras que en el segundo cuadrante o lado izquierdo del fotograma, la velocidad angular tiene valores positivos. Los valores $\omega_{zmax} = 1.0\,[rad/s]$ y $\omega_{zmin} = -1.0\,[rad/s]$ son los límites de velocidad angular del cuadricóptero.

$$\omega_{angularz} = \frac{\theta}{\pi/2} \left[\frac{rad}{s}\right] \tag{5.10}$$

5.8. Situaciones de detección

Los dos escenarios que se pueden presentar en la detección del objeto son los siguientes:

No hay detección: Si el carro de radio control no es detectado por el algoritmo de visión o no se encuentra en el campo de visión de la cámara, el robot cuadricóptero permanecerá quieto, de forma similar ocurre si el cuadricóptero pierde el carro de radio control. Se enviarán valores de velocidad lineal ($V_{linearx} = 0$, $V_{lineary} = 0$), velocidad angular ($\omega_{angularz} = 0$) y el robot aéreo se detendrá manteniendo su altitud ($V_{linearz}$).

Detección del objeto móvil: Si el carro de radio control es detectado, el algoritmo envía valores de velocidad lineal ($V_{linearx}$, $V_{lineary}$) y angular ($\omega_{angularz}$) para que el cuadricóptero lo siga. Los modelos presentados están entrenados con un *dataset* personalizado, con el fin de detectar la instancia en particular (`'RC'`), por lo anterior, aunque en el escenario se encuentre presente más objetos con características similares, solo será detectado el modelo de carro de radio control seleccionado. El algoritmo también está configurado para detectar una sola instancia del objeto que se desea detectar (ver Figura 5.6).

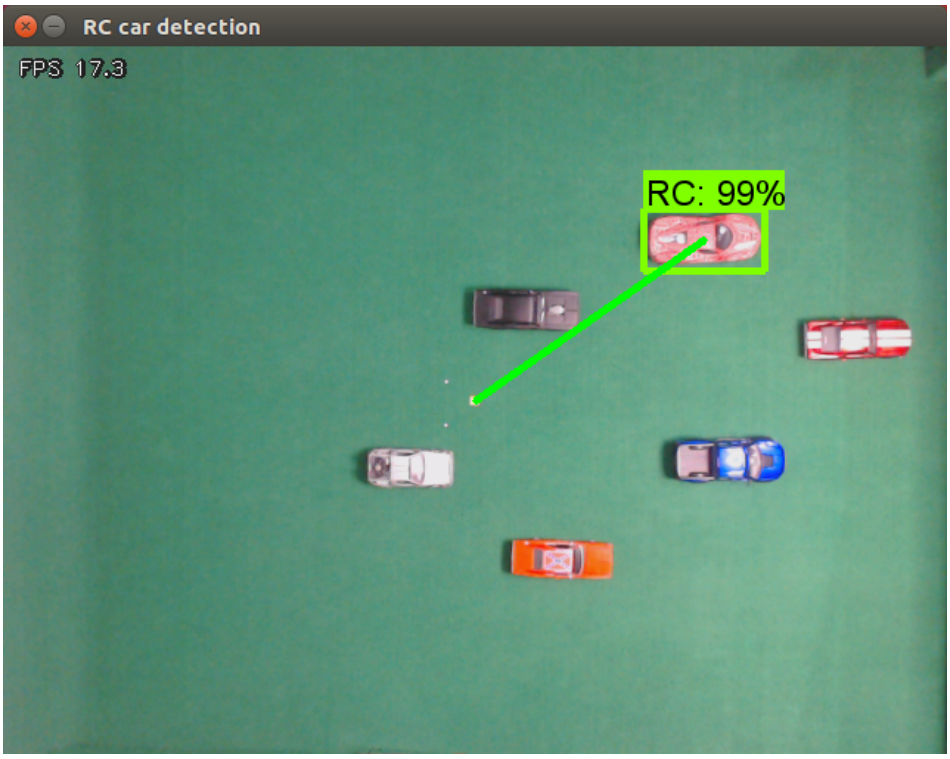


Figura 5.6: Representación de detección del carro de radio control. Imagen de salida del detector de objetos con el rendimiento FPS y porcentaje de precisión.

6 Interfaz de comunicación

En este capítulo se describe el funcionamiento de la interfaz de comunicación implementada en ROS, a través de nodos y *topics* que permiten intercambiar información entre el módulo Jetson TX2 y el modelo 3D del cuadricóptero. Uno de los requerimientos de este sistema es el uso de ROS para establecer la interfaz de comunicación entre el controlador del cuadricóptero y la unidad embebida de procesamiento de imágenes. Como se mencionó anteriormente, uno de los lenguajes de programación soportados por ROS es Python, por lo tanto, se utilizó este lenguaje para el procesamiento de imágenes y para los *scripts* en ROS.

6.1. Nodos y *topics* de la aplicación

Con la ayuda de `rqt_graph`, paquete que proporciona un complemento para visualizar los componentes ROS, se obtuvo una representación gráfica como en la Figura 6.1, donde se muestran los nodos que intervienen en el sistema y que se ejecutan en cada máquina. Estos nodos publican *topics* que se ubican encima de las flechas que apuntan a otros nodos que interactúan en la aplicación robótica.

En la Figura 6.1, se muestran los nodos y *topics* que se ejecutan en Jetson TX2 y en la Laptop Dell Precision M4500. A continuación, se describe la función de cada nodo:

- **talker:** Este nodo tiene como función publicar los valores de velocidades lineales y angulares, utilizando un mensaje de tipo `geometry_msgs/Twist.msg`. Este mensaje se envía a la Laptop Dell Precision M4500, a través del *topic* `\cmd_vel`. Este nodo también es el encargado de despegar y controlar el vuelo del cuadricóptero.
- **gazebo:** Este es el nodo encargado de lanzar el ambiente exterior (*outdoor*) de simulación en Gazebo y genera un modelo 3D del *hector quadrotor* configurado con sensores entre ellos el Hokuyo UTM-30LX, este es un escáner láser 2D pequeño, preciso y con alta velocidad de detección de obstáculos. También se inicia un nodo RViz con parámetros de control para el vuelo del cuadricóptero.

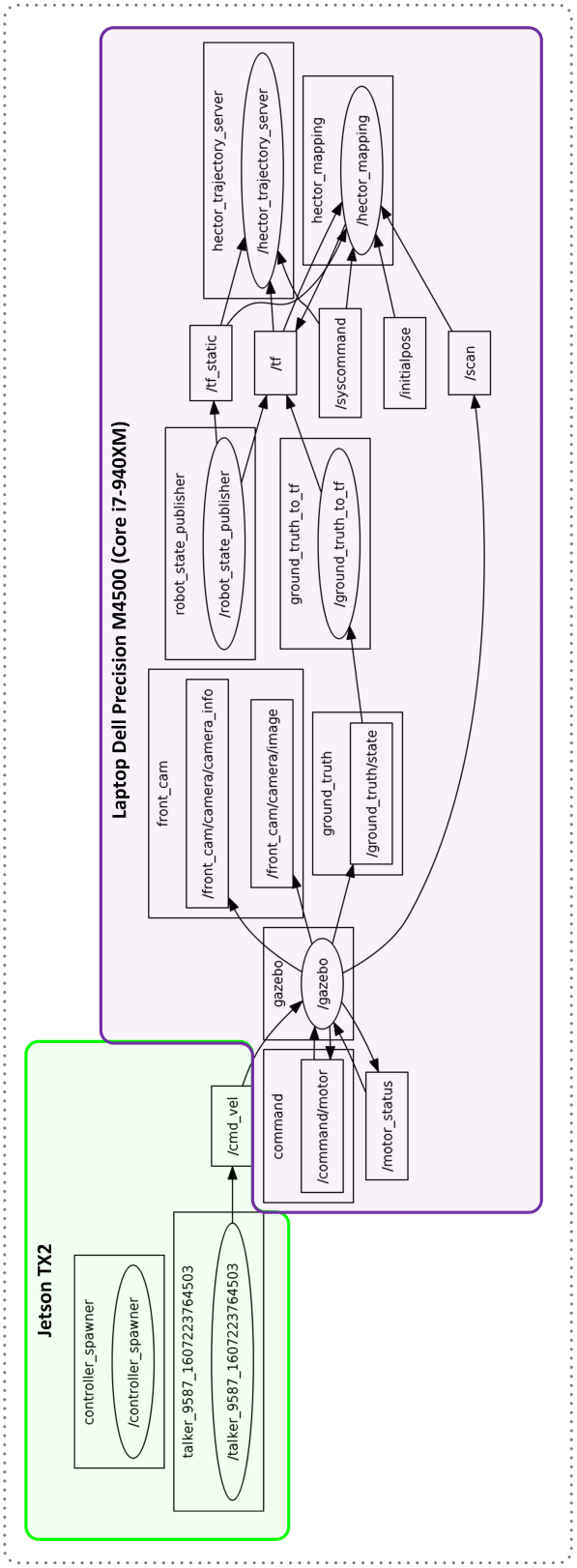


Figura 6.1: Gráfica de nodos y *topics* ROS activos en el sistema.

6.2. Ejecución de ROS en múltiples máquinas

Esta aplicación robótica requiere la comunicación entre dos máquinas, en este caso entre el módulo Jetson TX2 y la Laptop Dell Precision M4500 donde se ejecuta el ambiente de simulación del modelo 3D del cuadricóptero con el metapaquete *hector quadrotor*. Para establecer la comunicación ROS entre múltiples máquinas, se debe ejecutar en una de ellas el nodo *master* ROS y conectar la otra máquina a este nodo a través de la misma red local.

La Tabla 6.1 presenta una lista de los *topics* ROS controlados y publicados desde Jetson TX2. Los valores publicados por cada *topic*, son las velocidades de entrada utilizadas para controlar el cuadricóptero.

***Topic* ROS**	**Descripción**
/cmd_vel/twist/linear/x	Control de movimiento sobre el eje x
/cmd_vel/twist/linear/y	Control de movimiento sobre el eje y
/cmd_vel/twist/linear/z	Control de movimiento sobre el eje z
/cmd_vel/twist/angular/x	Control de rotación alrededor del eje x
/cmd_vel/twist/angular/y	Control de rotación alrededor del eje y
/cmd_vel/twist/angular/z	Control de rotación alrededor del eje z

Tabla 6.1: *Topics* ROS controlados en Jetson TX2.

En la Tabla 6.2 se muestran algunos de los *topics* ROS a los que se suscribe el cuadricóptero *hector quadrotor*. Esta lista contiene solo aquellos *topics* utilizados para mostrar los resultados y la validación del sistema que se presentan en el Capítulo 7.

***Topic* ROS**	**Descripción**
/ground_truth/state/twist/twist/linear/x	Movimiento cuadricóptero sobre el eje x
/ground_truth/state/twist/twist/linear/y	Movimiento cuadricóptero sobre el eje y
/ground_truth/state/twist/twist/linear/z	Movimiento cuadricóptero sobre el eje z
/ground_truth/state/twist/twist/angular/z	Rotación del cuadricóptero alrededor del eje z
/ground_truth/state/pose/pose/position/x	Pose del cuadricóptero en el eje x
/ground_truth/state/pose/pose/position/y	Pose del cuadricóptero en el eje y

Tabla 6.2: Suscripción a *topics* ROS desde el cuadricóptero *hector quadrotor*.

7 Resultados y contribución

En este capítulo se muestran los diferentes experimentos y misiones realizadas para validar el funcionamiento del sistema de visión en la detección y seguimiento del carro de radio control por parte del cuadricóptero. Adicional, se muestran los resultados de evaluar el rendimiento en FPS y la precisión (mAP) de los modelos en la tarea de detección del carro de radio control con etiqueta de clase `'RC'`.

7.1. Condiciones experimentales

La recopilación los resultados de evaluación de métricas de rendimiento y precisión de los modelos, incluyó un total de 1000 iteraciones en el algoritmo. Terminada cada prueba o misión, se exportan los resultados en un archivo CSV y se calculan los promedios. Los datos son tabulados y presentados a través gráficas que permiten comparar los resultados obtenidos. La validación de la detección y el seguimiento del objetivo móvil, se evidencian a través de diferentes gráficas que comparan los valores de velocidad lineal y velocidad angular de entrada, con respecto a los valores del cuadricóptero *hector quadrotor*.

Existen algunas consideraciones a tener en cuenta para determinar las condiciones de funcionamiento del sistema de visión. La altura máxima en la que se detecta el carro de radio control de forma constante es de 2 $[m]$, esta es la distancia en la que se encuentra ubicada la cámara con respecto del suelo.

Durante las simulaciones, el carro de radio control se ubicó en el campo de visión superior de la cámara, justo arriba de su centro de proyección en el plano de imagen. Los movimientos aleatorios y la posición del carro de radio control se limitan a la zona sombreada (verde), como se observa en la Figura 5.5.

7.2. Resultados de rendimiento

En esta sección, se compara el desempeño en fotogramas por segundo (FPS) obtenido por cada modelo implementado en Jetson TX2. Los resultados se calcularon con el mismo escenario y ubicación del objetivo móvil terrestre, y con la misma tarea de detectar el carro de radio control.

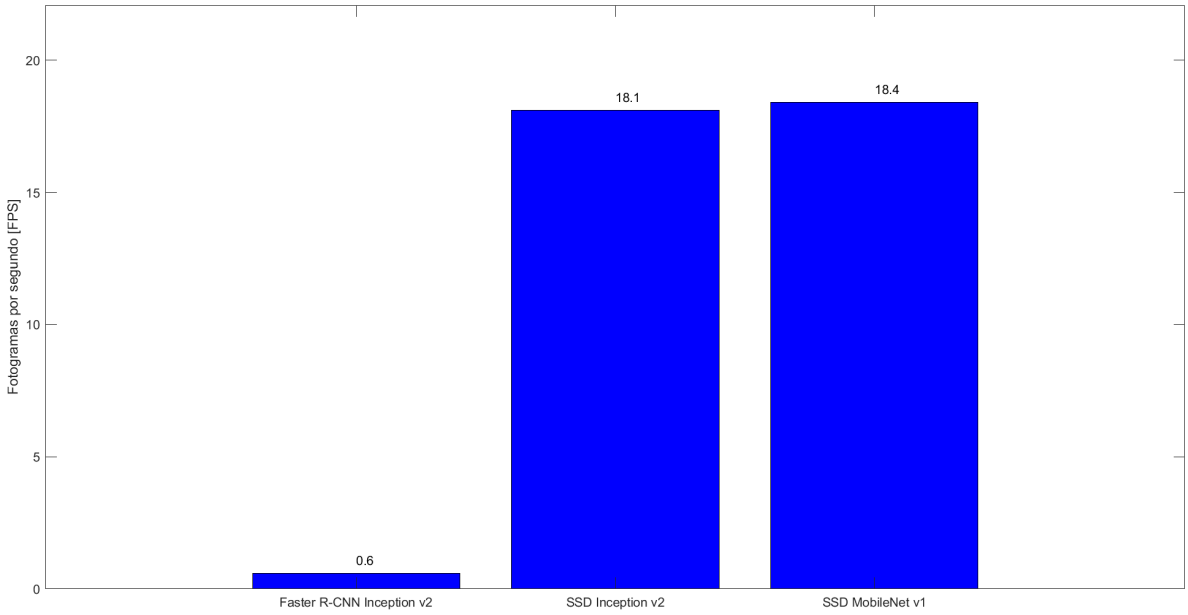


Figura 7.1: Rendimiento en FPS de modelos detectores de objetos: Faster R-CNN Inception v2, SSD Inception v2, SSD MobileNet v1 en Jetson TX2.

La Figura 7.1 y la Tabla 7.1 muestran los valores del rendimiento en FPS de los modelos detectores de objetos implementados para una resolución de entrada de 640×480 [$pixeles$]. Los dos detectores SSD Inception v2 y SSD MobileNet v1, son significativamente más rápidos que Faster R-CNN. La velocidad de fotogramas de los modelos SSD ronda entre $18.1 \sim 18.4$ FPS, mientras que en Faster R-CNN Inception v2 es de alrededor de 0.6 FPS.

Modelo	**Resolución de entrada [px]**	**Fotogramas [FPS]**
Faster R-CNN Inception v2	640×480	0.6
SSD Inception v2	640×480	18.1
SSD MobileNet v1	640×480	18.4

Tabla 7.1: Rendimiento de modelos *deep learning* en fotogramas por segundo (FPS) en Jetson TX2.

7.3. Resultados de precisión

En este apartado, se presentan los valores de precisión mAP[%] de los modelos detectores de objetos para la clase `'RC'` del carro de radio control. La Tabla 7.2 muestra que el modelo más preciso es Faster R-CNN Inception v2 con un valor mAP de 100 %, sin embargo, como se mencionó anteriormente, también es el más lento en rendimiento FPS. Le sigue SSD Inception v2 con 94.44 %, y finalmente SSD MobileNet v1 con una precisión de 84.97 %.

Modelo	**Fotogramas [FPS]**	**mAP [%]**
Faster R-CNN Inception v2	0.6	100
SSD Inception v2	18.1	94.44
SSD MobileNet v1	18.4	84.97

Tabla 7.2: Precisión mAP[%] y rendimiento en FPS de los modelos en Jetson TX2.

Las Figuras 7.3, 7.4, 7.2, corresponden a imágenes de salida de los detectores de objetos Faster R-CNN Inception v2, SSD Inception v2 y SSD MobileNet v1, respectivamente. Cada captura muestra el *bounding box* sobre el carro de radio control, la etiqueta, el porcentaje de precisión y la velocidad en FPS.

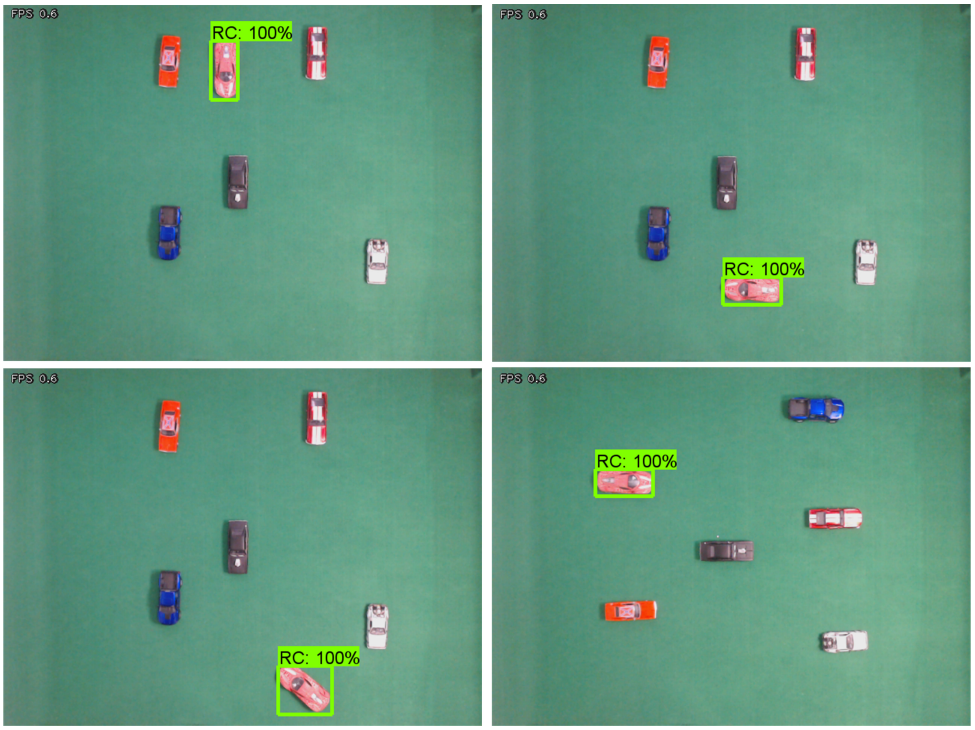


Figura 7.2: Imágenes de salida de Faster R-CNN Inception v2.

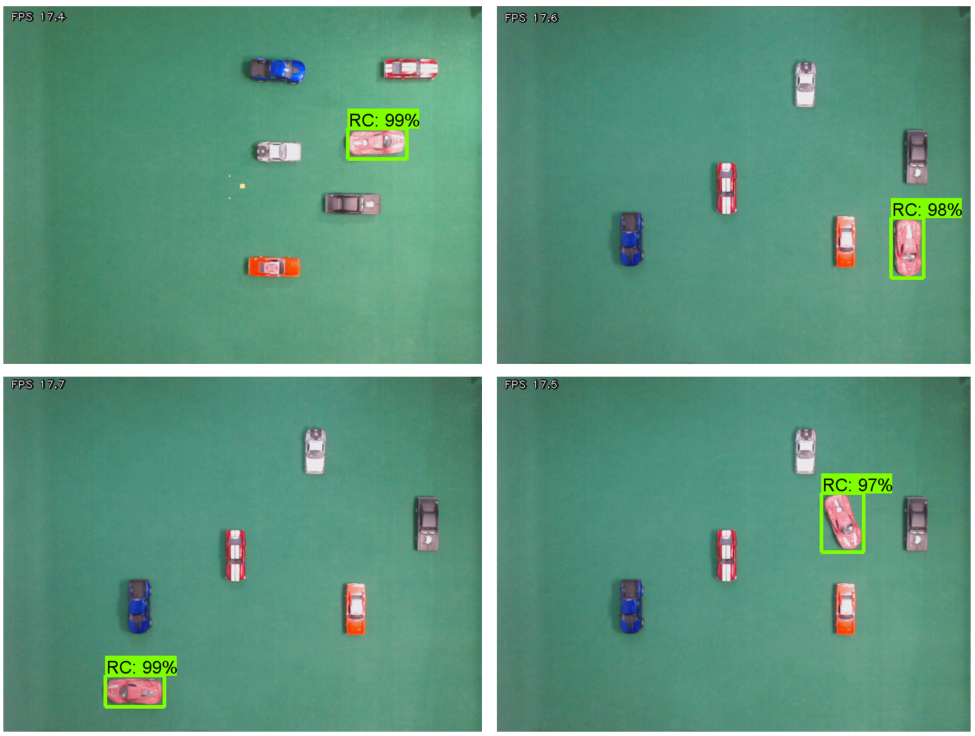


Figura 7.3: Imágenes de salida de SSD Inception v2.

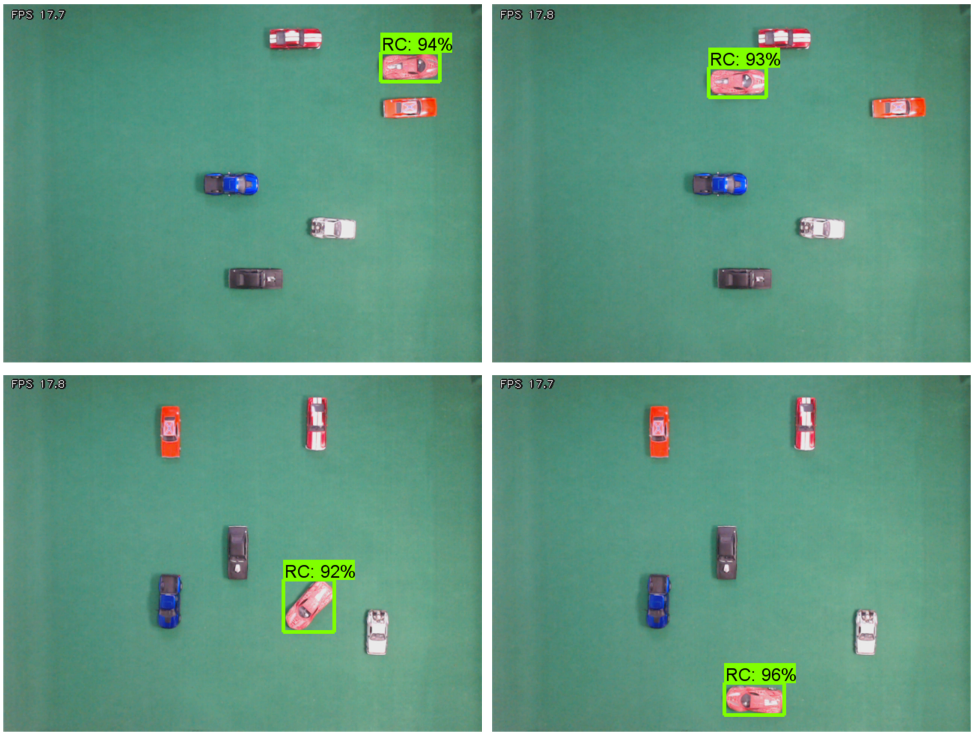


Figura 7.4: Imágenes de salida de SSD MobileNet v1.

7.4. Resumen de resultados

La Tabla 7.3 recopila los resultados obtenidos por cada modelo en las métricas de evaluación consideradas. Para cada modelo se evaluó, el rendimiento en FPS, la precisión en mAP(%) y la velocidad máxima a la que puede desplazarse el carro de radio control para que pueda ser detectado.

Modelo	**Fotogramas [FPS]**	**mAP [%]**	**Vel. máx. RC [m/s]**
Faster R-CNN Inception v2	0.6	100	0.5
SSD Inception v2	18.1	94.44	15.3
SSD MobileNet v1	18.4	84.97	16.1

Tabla 7.3: Resumen de resultados de detectores de objetos. Rendimiento en FPS y precisión mAP. Velocidad máxima del carro de radio control.

La Figura 7.5 presenta un compendio con la imágenes de salida obtenida por cada modelo en el procesamiento de la misma imagen de entrada.

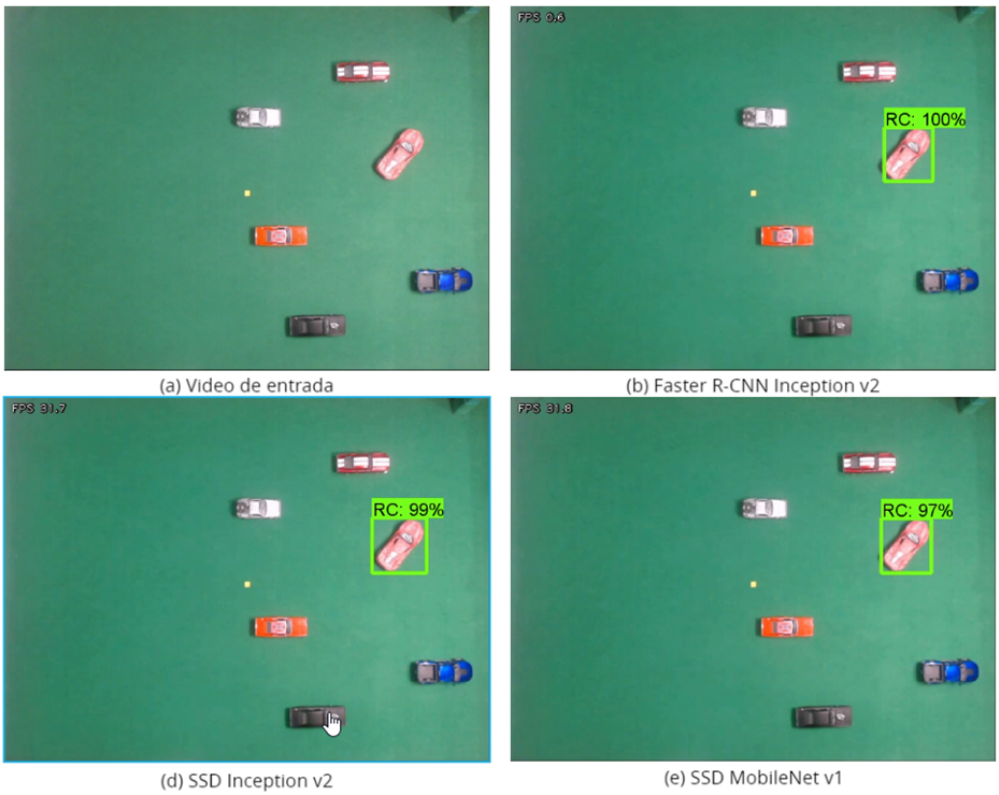


Figura 7.5: Imágenes de salida de detectores de objetos para la misma imagen de entrada.

7.5. Escenario I

En el primer escenario, el carro de radio control se ubica justo en frente del cuadricóptero y se mueve a velocidad constante siguiendo una trayectoria rectilínea. Luego que despega, el cuadricóptero alcanza los valores de velocidad lineal en X de $V_x \approx 0.45\,[m/s]$, velocidad lineal en Y de $V_y \approx 0.0\,[m/s]$ y velocidad angular en Z de $\omega_z \approx 0.0\,[rad/s]$, valores que corresponden con los deseados. En cuanto a su pose XY, el cuadricóptero sigue una trayectoria rectilínea con una posición XY de $X \approx 6\,[m]$, $Y \approx 0\,[m]$, como se observa en la Figura 7.6.

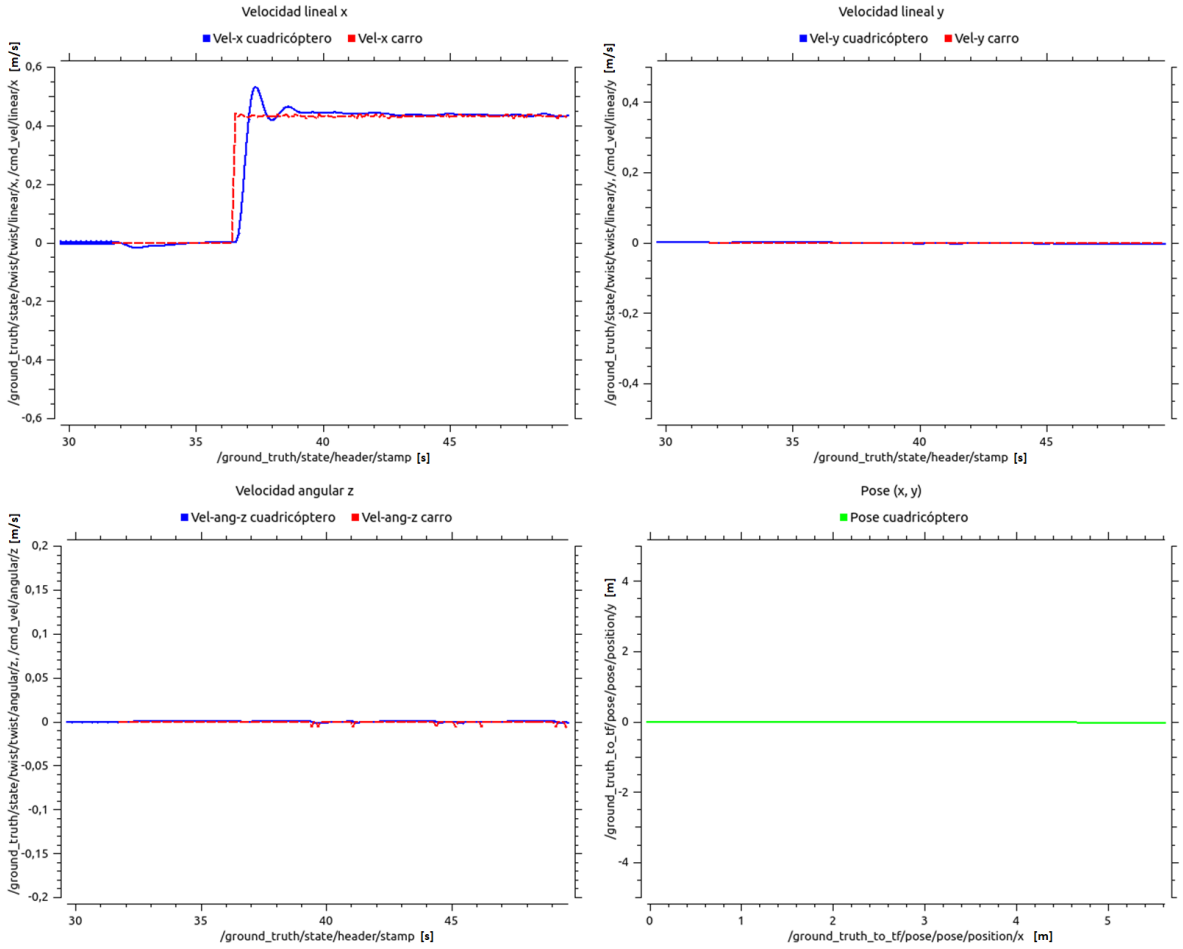


Figura 7.6: **Misión 1a:** Valores de velocidad lineal (V_x, V_y) y angular (ω_z) de entrada y salida. Valores de velocidad de entrada controlados por Jetson TX2 (rojo). Valores de velocidad del cuadricóptero *hector quadrotor* (azul). Pose del cuadricóptero *hector quadrotor* (verde).

Los resultados que se muestran en la Figura 7.7 pertenecen a la misma misión presentada en la Figura 7.6 transcurrido un instante de tiempo. Como se puede observar, los valores de velocidad lineal en X de $V_x \approx 0.45\,[m/s]$, velocidad en Y de $V_y \approx 0.0\,[m/s]$ y velocidad angular en Z de $\omega_z \approx 0.0\,[rad/s]$ se mantienen, como también la trayectoria rectilínea del cuadricóptero con una posición XY con $X \approx 27\,[m]$, $Y \approx 0.2\,[m]$.

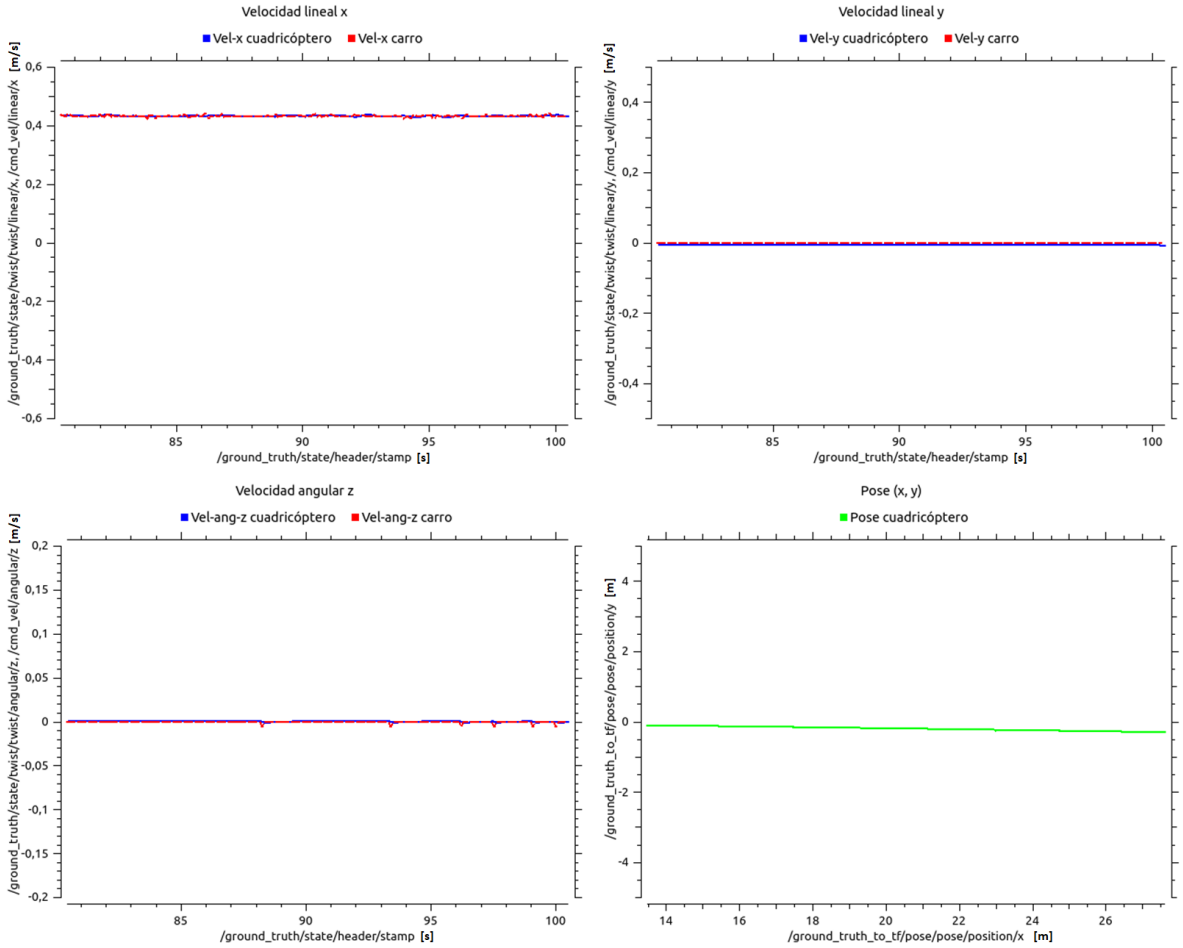


Figura 7.7: **Misión 1b:** Valores de velocidad lineal (V_x, V_y) y angular (ω_z) de entrada y salida. Valores de velocidad de entrada controlados por Jetson TX2 (rojo). Valores de velocidad del cuadricóptero *hector quadrotor* (azul). Pose del cuadricóptero *hector quadrotor* (verde).

7.6. Escenario II

En el segundo escenario, el carro de radio control se mueve realizando maniobras consecutivas y aleatorias en la zona de cobertura de búsqueda, entre el primer y segundo cuadrante (ver Figura 5.5), cambiando permanentemente el ángulo del centroide con respecto al centro de la imagen (θ).

La Figura 7.8 muestra que el cuadricóptero alcanza los valores de velocidad lineal y de velocidad angular deseados. El valor de velocidad lineal en X, oscila entre $V_x = (0.2\,[m/s]\,,\, 0.6\,[m/s])$, la velocidad lineal en Y entre $V_y = (-0.3\,[m/s]\,,\, 0.4\,[m/s])$ y la velocidad angular en Z entre $\omega_z = (-0.4\,[rad/s]\,,\, 0.4\,[rad/s])$. La traslación del cuadricóptero en XY muestra una trayectoria oscilatoria en X que llega a $X \approx 7\,[m]$.

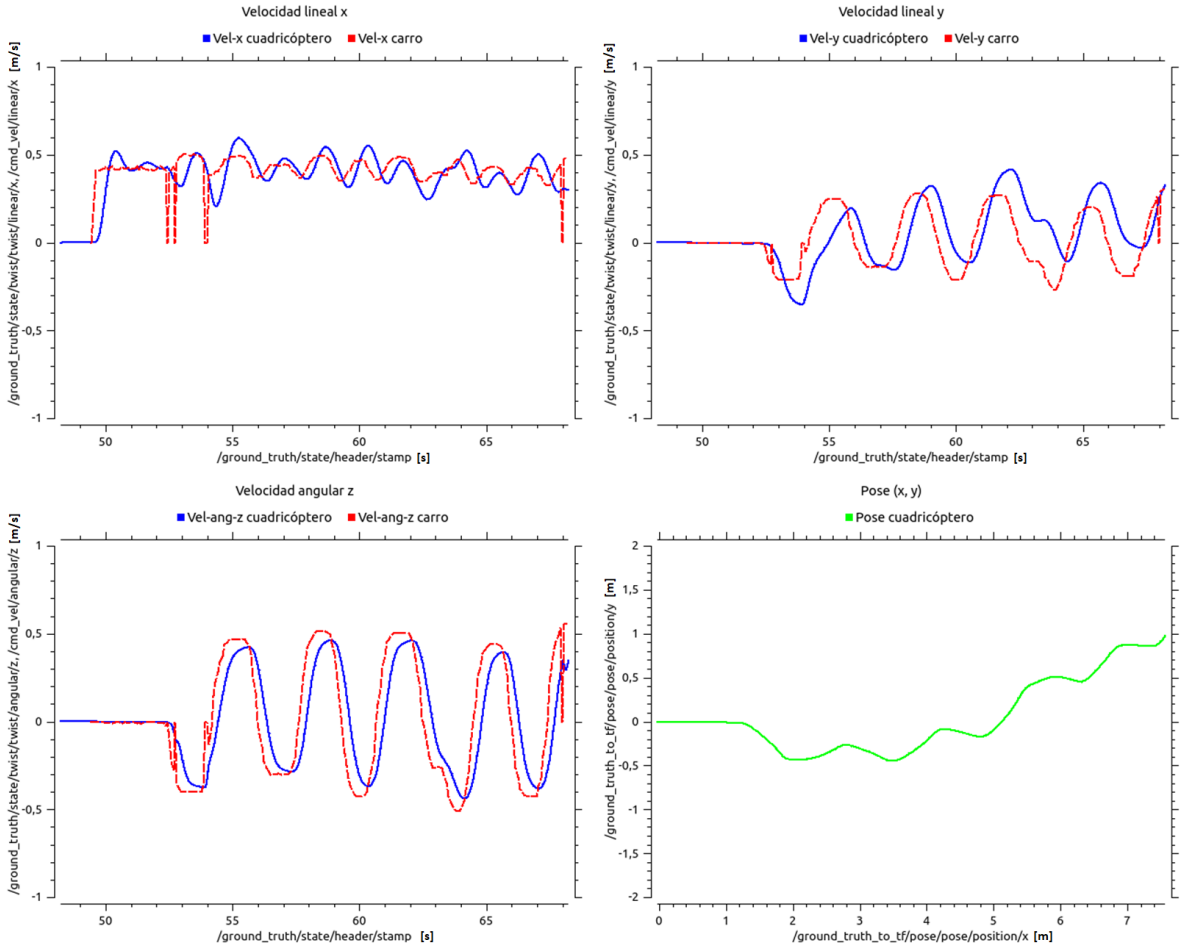


Figura 7.8: **Misión 2:** Valores de velocidad lineal (V_x, V_y) y angular (ω_z) de entrada y salida. Valores de velocidad de entrada controlados por Jetson TX2 (rojo). Valores de velocidad del cuadricóptero *hector quadrotor* (azul). Pose del cuadricóptero *hector quadrotor* (verde).

Por último, se presenta otra misión en el escenario objetivo móvil. La Figura 7.9 recopila los resultados obtenidos en este experimento. Como se muestra en la figura, el valor de velocidad lineal en X oscila entre $V_x = (0.3\,[m/s]\,,\, 0.6\,[m/s])$, la velocidad lineal en Y entre $V_y = (-0.3\,[m/s]\,,\, 0.4\,[m/s])$ y la velocidad angular en Z entre $\omega_z = (-0.5\,[rad/s]\,,\, 0.5\,[rad/s])$. La traslación del cuadricóptero en XY muestra una trayectoria oscilatoria en X de $x \approx 12\,[m]$.

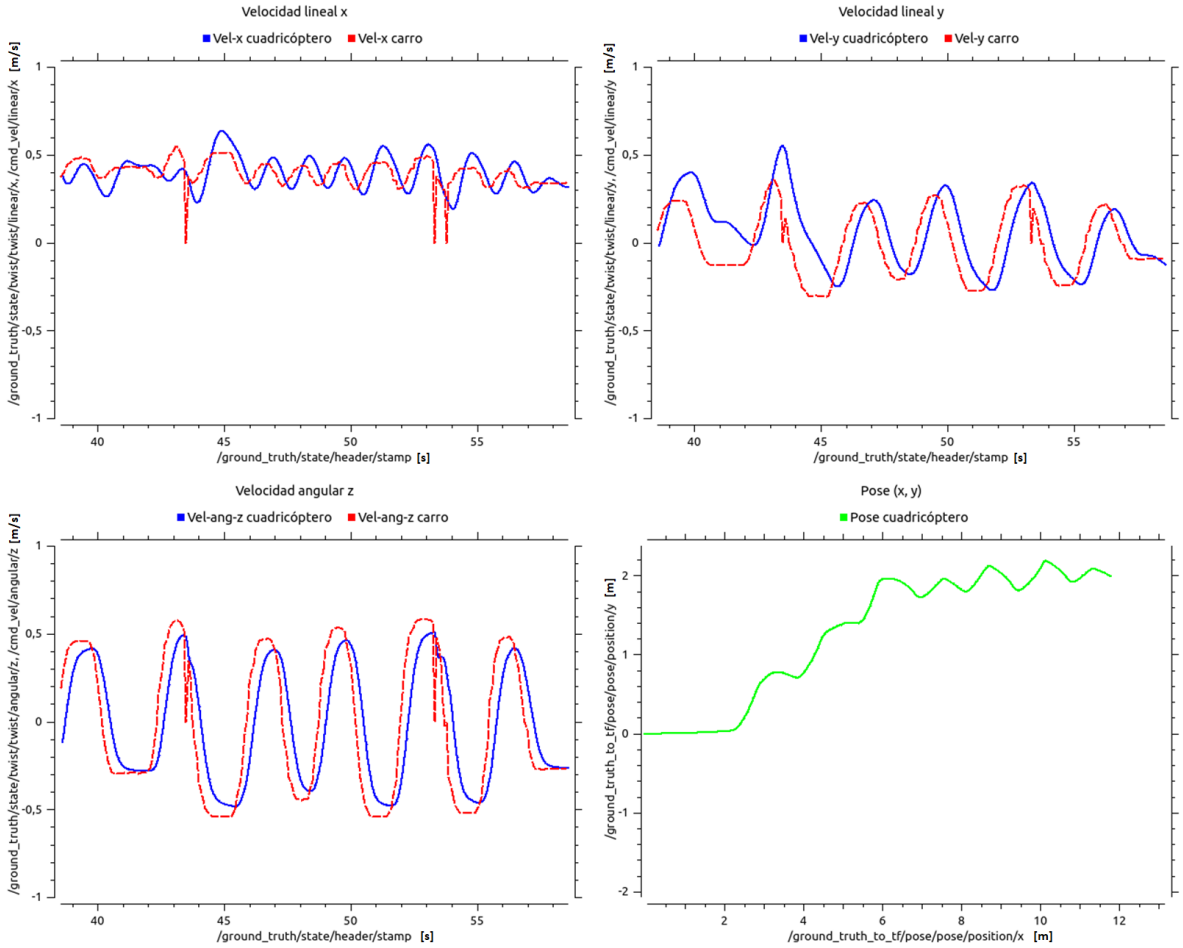


Figura 7.9: **Misión 3:** Valores de velocidad lineal (V_x, V_y) y angular (ω_z) de entrada y salida. Valores de velocidad de entrada controlados por Jetson TX2 (rojo). Valores de velocidad del cuadricóptero *hector quadrotor* (azul). Pose del cuadricóptero *hector quadrotor* (verde).

8 Discusión y conclusiones

El desarrollo de sistemas robóticos autónomos se ha incrementado en los últimos años, lo que antes era exclusividad de algunas instituciones, hoy en día se ha convertido en una tendencia global impulsada por investigadores, comunidades científicas y empresas en el mundo. En particular, los vehículos aéreos no tripulados (UAV) tienen tareas habituales de inspección, vigilancia y seguimiento de objetos ubicados en tierra.

La principal contribución de este trabajo de grado es presentar un sistema de visión *deep learning* en tiempo real, que puede ir a bordo de un UAV, con el fin de dotarlo con la capacidad de detectar y seguir de forma autónoma un objeto en movimiento. Los sensores ópticos han demostrado ser una gran fuente de información para la navegación de plataformas UAV. Procesar esta información visual a través de algoritmos y técnicas de visión por computador, brindan mayor autonomía a un UAV para la ejecución de sus tareas.

Los modelos *deep learning* resuelven varios de los problemas asociados con la detección de objetos. En este trabajo se han entrenado e implementado detectores de objetos y algoritmos capaces de detectar y seguir un carro de radio control. En el entrenamiento de detectores *deep learning* es esencial conformar un amplio *dataset* de imágenes con grandes variaciones en escala, fondo e iluminación, ya que de los datos de entrada va a depender el correcto entrenamiento y funcionamiento de los rastreadores de objetos. Las técnicas de optimización adoptadas, permiten la ejecución de las inferencias de modelos *deep learning* sobre el módulo embebido Jetson TX2. El uso de TensorRT permitió optimizar las inferencias de los modelos TensorFlow resultantes con una precisión de FP16 (punto flotante de 16 bits), lo que representó un mejor rendimiento sin que se viera afectada la precisión de los modelos.

Para evitar perder el objetivo móvil se debe incrementar al máximo el rendimiento del detector de objetos. La configuración NVPModel en el modo MAX-N, evidenció que es posible ejecutar inferencias *deep learning* casi hasta dos veces más rápido que con el modo predeterminado MAX-P. Los detectores SSD presentan rendimientos entre 9.6 $\sim$ 9.7 FPS en el modo MAX-P, mientras que en MAX-N se alcanzan rendimientos de 18.1 $\sim$ 18.4 FPS. Las mediciones de potencia y corriente durante la ejecución de los modelos detectores de objetos en el modo MAX-N muestran un leve aumento de 4.3 $[W]$ y 0.3 $[A]$, respectivamente. Estos resultados representan un punto de partida para el diseño del módulo de energía que alimente

el *hardware* del sistema. Para la tarea de detección y seguimiento del carro de radio control, se obtuvo como resultado que los detectores de objetos *Single Shot Multibox Detector* (SSD) son los que presentan el mejor balance de rendimiento y precisión, en comparación con Faster R-CNN, que aunque es más preciso, ofrece un bajo rendimiento.

La arquitectura *hardware-in-the-loop* (HIL) presentada en este trabajo de grado ofrece una solución híbrida funcional que incluye el *hardware* real diseñado y configurado para ir a bordo de un UAV, como es el caso de Jetson TX2, que proporciona una solución eficiente y de alto rendimiento en la optimización y ejecución de las inferencias de modelos TensorFlow seleccionados y entrenados para la detección del carro de radio control. Por su parte, la simulación HIL utiliza ROS, RViz y Gazebo para integrar una plataforma UAV. La plataforma aérea es provista por el metapaquete *hector quadrotor* que proporciona un modelo en 3D de un cuadricóptero e incluye sensores de altitud, posición y velocidad con los que puede navegar en un entorno externo libre de obstáculos. Esta arquitectura representa una posibilidad realista de implementación de la aplicación robótica autónoma para un UAV, su utilización dispone de varias ventajas, en primer lugar, la puesta en marcha del sistema disminuye significativamente los costos y permite realizar múltiples experimentos sin comprometer la seguridad de la plataforma aérea ni del entorno.

Las simulaciones y diversos experimentos realizados para la validación del sistema, han resultado satisfactorios. El cuadricóptero alcanza los valores de velocidad deseados y sigue la trayectoria esperada de acuerdo con la ubicación y desplazamiento del objetivo móvil terrestre. La metodología propuesta y el sistema de visión por computador embebido desarrollado para detectar y rastrear un carro de radio control desde un UAV, podría ser ajustado y extendido para la detección de otro tipo de objetivos, siempre que se cumplan las condiciones para su correcto funcionamiento. Por ejemplo, las medidas del carro de radio control detectado en este proyecto, cuenta con dimensiones: 10 $[cm]$ de ancho, 23 $[cm]$ de largo y 6.5 $[cm]$ de alto, y la altura máxima máxima para la detección de este objeto que es es de 2 $[m]$. Haciendo una extrapolación de este resultado, se podría detectar un vehículo compacto con dimensiones 1.6 $[m]$ de ancho, 3.8 $[m]$ de largo y 1.5 $[m]$ de alto, a una altura máxima de detección de ~ 32 $[m]$. Teniendo en cuenta, las mismas consideraciones y resolución de imagen de entrada.

8.1. Trabajos futuros

Durante el desarrollo de este trabajo de grado se han identificado diversas funcionalidades, mejoras y trabajos futuros derivados de este proyecto. A continuación, se proponen tres áreas de trabajo y se sugieren desarrollos para continuar o mejorar el sistema presentado:

1. Sustituir el entorno de simulación del UAV por el *hardware* y componentes necesarios para llevar el sistema de visión a una aplicación con un UAV real.
2. Desarrollar algoritmos para la estimación del viento y control de vuelos UAV, con la finalidad de mantener estable el vuelo en condiciones atmosféricas adversas como las corrientes de viento.
3. Incluir otras categorías de objetos a detectar en el entrenamiento de los modelos *deep learning*, con el objetivo de extender la aplicabilidad y tareas que puede realizar el UAV.

A

Calibración de cámara

La calibración de una cámara permite determinar los coeficientes de distorsión, parámetros intrínsecos y extrínsecos de la misma. Este proceso se puede aproximar al modelo de cámara estenopeica, que consiste en estimar una relación geométrica entre un punto 3D del mundo real y su proyección 2D correspondiente en un plano de imagen. Para obtener estas correspondencias se utilizan una serie de fotografías de un patrón de calibración, como un tablero de ajedrez, luego de hallar los parámetros de la cámara es posible evaluar su precisión o corregir la distorsión de la lente.

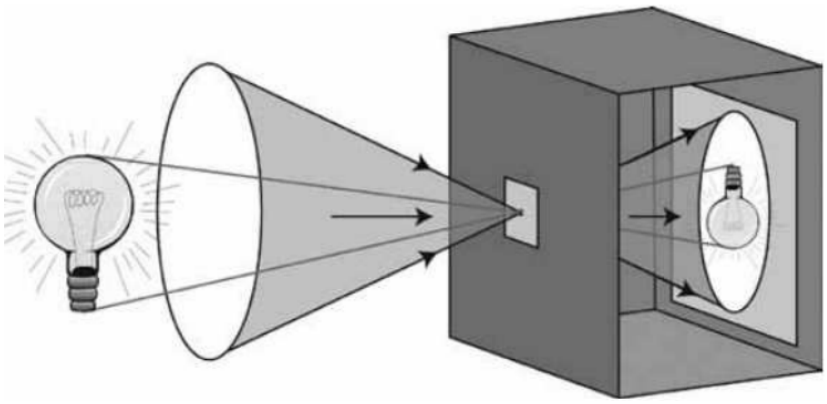


Figura A.1: Modelo de cámara estenopeica. Tomado de [125].

Las cámaras de bajo costo suelen introducir distorsiones radiales significativas a las imágenes capturadas. Este tipo de distorsión hace que líneas rectas de un escenario real, sean representadas en las imágenes como curvas. Este problema se resuelve de la siguiente manera:

$$x_{\text{corrección}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{A.1}$$

$$y_{\text{corrección}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{A.2}$$

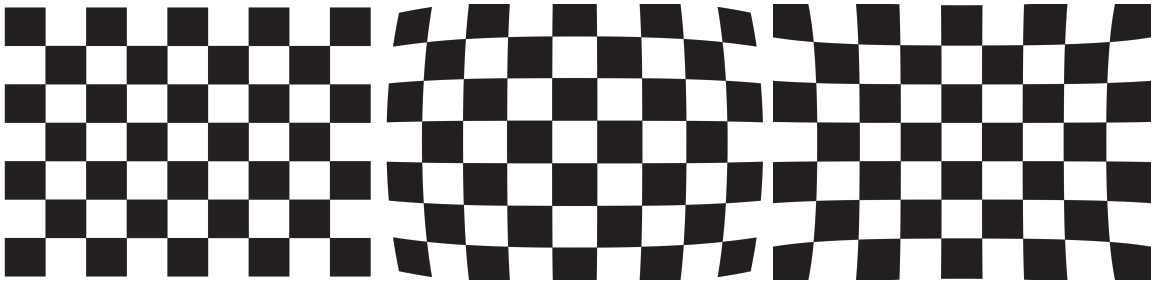
$$Coeficientes\ de\ distorsión = (k_1, k_2, p_1, p_2, k_3) \tag{A.3}$$

Los parámetros intrínsecos son únicos de una cámara e incluyen la distancia focal ($f_x,\ f_y$) y el centro óptico ($c_x,\ c_y$).

$$Matriz\ de\ cámara = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{A.4}$$

OpenCV posee librerías con funciones específicas para la calibración de cámaras. Las funciones de calibración permiten calcular los parámetros intrínsecos y extrínsecos de la cámara que se desea utilizar.

El procedimiento para la calibración de la cámara consiste en seleccionar un patrón, en este caso una cuadrícula o rejilla similar a un tablero de ajedrez.



(a) (b) (c)

Figura A.2: Distorsión de imagen radial; **a:** Imagen sin distorsión, **b:** Distorsión barril (distorsión radial positiva), **c:** Distorsión cojín (distorsión radial negativa).

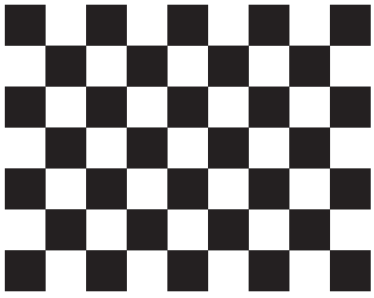


Figura A.3: Patrón de calibración de cámara.

Luego, se capturan un conjunto de imágenes del tablero desde diferentes puntos de vista, distancias y orientaciones, de tal manera que se reúna la suficiente información para calcular correctamente la matriz de la cámara y los coeficientes de distorsión. El patrón seleccionado debe fijarse sobre una superficie plana para que no se vea disminuida la precesión de la calibración de la cámara. La documentación oficial de OpenCV proporciona un código fuente

general en Python para la calibración de cámaras, éste puede modificarse de acuerdo a las dimensiones y cantidad de cuadros de la rejilla. La función `cv2.calibrateCamera`.

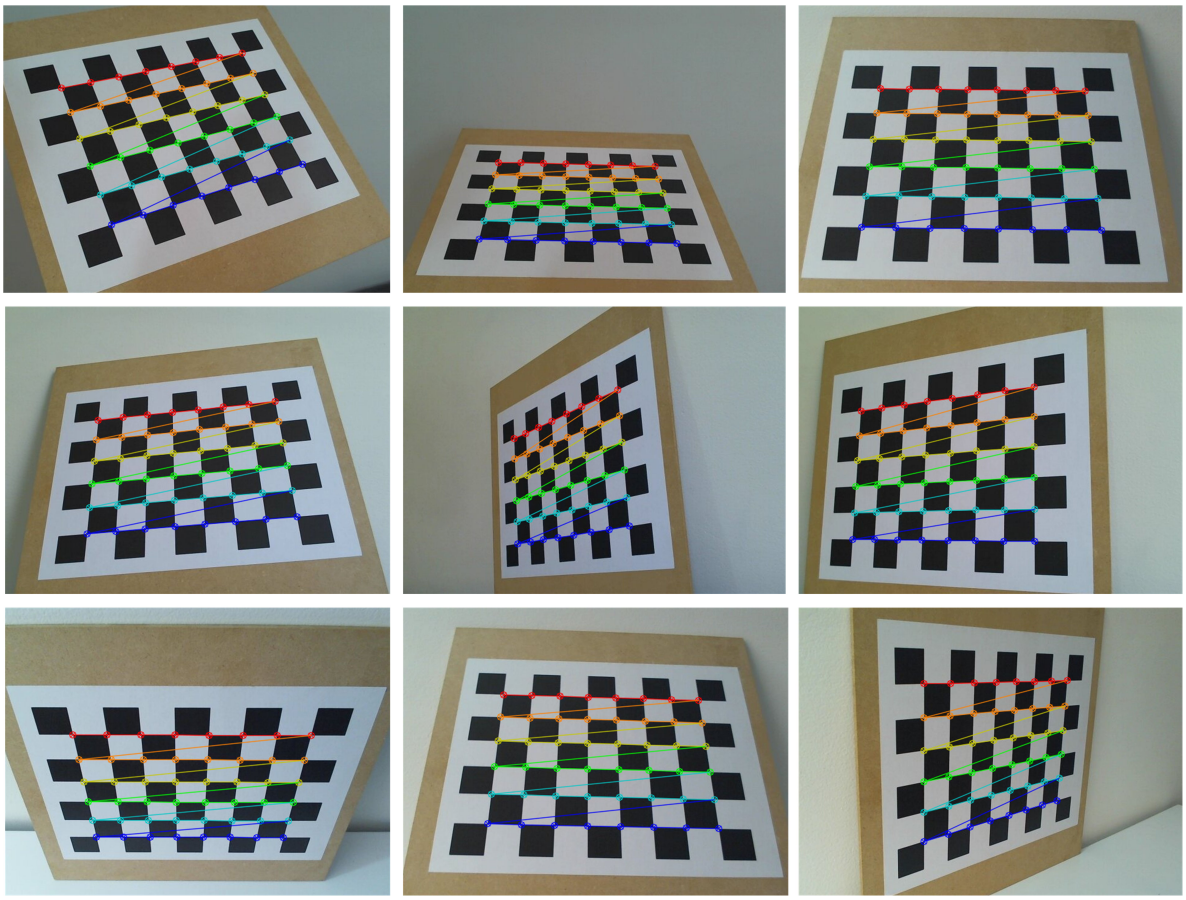


Figura A.4: Imágenes resultantes de la calibración de cámara web con las funciones de OpenCV.

A.1. Parámetros intrínsecos de la cámara

$$Mtx = \begin{bmatrix} 727.80550575 & 0 & 294.51660499 \\ 0 & 730.3975178 & 285.03854064 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.5}$$

$$dist = \begin{bmatrix} -8.80e-03 & 2.65e-01 & 5.22e-04 & -8.84e-04 & -1.98e+00 \end{bmatrix} \tag{A.6}$$

$$Total\ error : 0.0189951131241 \tag{A.7}$$

B

ROS (*Robot Operating System*)

A mediados del 2000, se acrecentó una necesidad común en medio de muchas personas de la comunidad de investigación en robótica, la de desarrollar un *framework* colaborativo que permitiera desarrollar piezas de *software* reutilizables y funcionales en varios robots haciendo pequeños cambios en el código.

ROS inicia como proyecto en la Universidad de Stanford con la creación de varios prototipos de sistemas de *software* flexibles y dinámicos destinados al uso de los robots. A finales del 2007, Willow Garage[48], una incubadora y laboratorio de investigación en robótica, destinó recursos significativos para extender los conceptos de este proyecto, lo que conllevó a que investigadores contribuyeran con su tiempo y experiencia para el desarrollo del núcleo y los paquetes de *software* fundamentales que componen ROS.

Aunque no es considerado un sistema operativo, ROS actúa de manera similar a uno, ya que proporciona al usuario abstracción de *hardware*, un conjunto de librerías, *drivers*, gestión de paquetes y herramientas de visualización que ayudan al desarrollo de aplicaciones robóticas de acuerdo con las necesidades de cada usuario.

El modelo de repositorio federado que sigue ROS, se ha convertido en una de sus principales fortalezas, ya que motiva a usuarios y desarrolladores a alojar sus repositorio de paquetes en ROS, sin perder la propiedad y el control sobre los mismos. La comunidad ROS sigue creciendo muy rápido, cada vez son más los usuarios y las empresas de robótica en el mundo, que están incorporando o migrando sus aplicaciones a ROS. Esta tendencia ha generado gran cantidad de oportunidades laborales en este campo.

Existen algunas características por las que se prefiere usar ROS sobre sobre otras plataformas robóticas como Orocos[49], YARP[50], Player[51], Carmen[52], MOOS[53], Orca[54] y Microsoft

[48] http://www.willowgarage.com/
[49] http://www.orocos.org/
[50] http://www.yarp.it/
[51] http://playerstage.sourceforge.net/
[52] http://carmen.sourceforge.net/intro.html
[53] http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php
[54] http://orca-robotics.sourceforge.net/

Robotics Studio[55]. ROS incluye capacidades (*capabalities*) listas para usar, como por ejemplo localización y mapeo simultáneos (*Simultaneous Localization And Mapping (SLAM)*), y localización de Monte Carlo (*Adaptive Monte Carlo Localization (AMCL)*), paquetes que son utilizados para la navegación autónoma de los robots.

El gran número de herramientas de código abierto para la depuración, visualización y simulación, como RViz y Gazebo. El soporte y la compatibilidad con sensores como dispositivos láser que usan el método LIDAR, cámaras de profundidad como Kinect[56] o las series de Intel RealSense D400[57] y actuadores con tecnología avanzada como lo son los servomotores Dynamixel[58]. Otra característica que diferencia a ROS de las demás plataformas robóticas es que permite la comunicación entre nodos que pueden ser programados en diferentes lenguajes como C++, C, Python o Java.

La modularidad y los métodos robustos que proporciona ROS, permite que el sistema continúe funcionando aun cuando existan fallas en uno de los nodos o también, reanuda el funcionamiento del mismo ante una falla en los sensores o actuadores. La comunidad de ROS se encuentra activa para brindar soporte y resolver las dudas de los usuarios.

La estructura de ROS, se compone tres niveles principales: nivel de sistemas de archivos, nivel de computación gráfica y nivel de comunidad.

B.0.1. Arquitectura ROS

ROS es un *framework* para la distribución de procesos (nodos) que permite a los ejecutables ser diseñados de manera individual y acoplarse de forma flexible en tiempo de ejecución. Estos procesos se agrupan en paquetes y metapaquetes, que pueden ser fácilmente compartidos y distribuidos.

Los conceptos fundamentales de ROS son: nodos, mensajes, *topics*, servicios y acciones. Los nodos son procesos que realizan cálculos. ROS está diseñado para ser modular en una escala fina (*fine-grained*): sistema que típicamente se compone de muchos nodos. A pesar de la importante fluidez y baja latencia que necesita el control de un robot, ROS no es un sistema operativo de tiempo real, aunque es posible integrarlo con código en tiempo real [126].

Los nodos se comunican entre sí mediante el paso de mensajes. Un mensaje es una estructura de datos, que comprende varios tipos estándar como enteros, flotantes, booleanos y arreglos. Los mensajes pueden incluir estructuras y vectores anidados arbitrariamente. Estos se enrutan a través de un sistema de transporte de tipo publicación-suscripción. Un nodo envía un mensaje mediante su publicación en un determinado *topic*.

Un *topic* es un nombre que se utiliza para identificar el contenido del mensaje. Un nodo que esté interesado en un determinado tipo de datos se suscribirá al *topic* correspondiente. Pueden haber múltiples publicadores y suscriptores concurrentes para un solo *topic*, y un único nodo puede publicar y suscribirse a múltiples *topics* [127]. Un nodo en ROS también

[55] https://www.microsoft.com/en-us/download/details.aspx?id=29081

[56] https://azure.microsoft.com/es-es/services/kinect-dk/

[57] https://www.intel.la/content/www/xl/es/architecture-and-technology/realsense-overview.html

[58] http://www.robotis.us/dynamixel/

puede ofrecer un servicio bajo un nombre. Este servicio se activa con el mensaje de solicitud enviado por el cliente y una vez procesado, envía la respectiva respuesta de regreso al cliente quien está a su espera [128].

En un servicio ROS el usuario implementa una interacción solicitud/respuesta entre dos nodos, sin embargo, si la respuesta tarda mucho tiempo o el servidor no ha terminado su trabajo, bloquea la aplicación principal mientras se espera hasta que finalice la acción solicitada. Una acción en ROS es similar a un servicio, con la diferencia que en este caso, el nodo cliente no bloquea la ejecución hasta que recibe la respuesta, sino que continua ejecutando otras instrucciones de código y, cuando recibe la respuesta, el cliente ejecuta la acción programada. Este comportamiento se denomina asíncrono, mientras que un servicio ROS, es de naturaleza síncrona, es decir, bloquea la ejecución del nodo hasta que se recibe la respuesta [129]. Los paquetes `actionlib` proporcionan una forma estándar de implementar este tipo de tareas. `actionlib` es muy utilizado en la navegación de brazos de robots y en la navegación de robots móviles [130].

B.0.2. Nivel de sistema de archivos

La Figura B.1 muestra cómo están organizados los archivos y carpetas en el sistema.

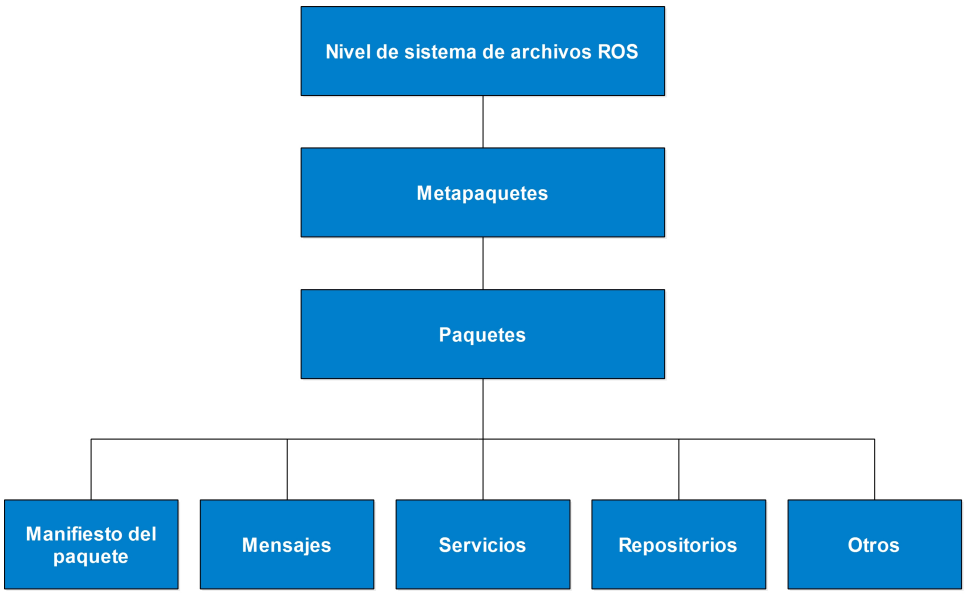


Figura B.1: Nivel de sistema de archivos en ROS. Tomado de [131].

- **Paquetes (*Packages*):** Representa la unidad más básica y fundamental de construcción en ROS. Los paquetes pueden contener procesos (nodos), bibliotecas y archivos de configuración que se organizan juntos en una sola unidad.
- **Metapaquetes (*Metapackages*):** Es el término que se usa para un grupo de paquetes que tienen una función especial. En anteriores distribuciones de ROS, se llamaban pilas

(*stacks*), sin embargo, luego se eliminó para darle paso a los metapaquetes.

- **Manifiesto del paquete y metapaquete:** Es un archivo llamado `package.xml` que se encuentra en la carpeta raíz de cada paquete compatible con `catkin` (Sistema de macros de CMake para el adecuado manejo de la compilación e instalación del paquete). Este archivo proporciona información del paquete, nombre, descripción para su uso, versión, autor, tipo de licencia, dependencias con otros paquetes, compilación y banderas.
- **Mensajes:** Son un tipo de información que se envía de un proceso (nodo) a otro. Es posible definir un mensaje personalizado dentro de un subdirectorio `msg` de un paquete. Los tipos de mensajes utilizan convenciones de nomenclatura estándar, nombre del paquete y nombre del archivo `.msg`. Por ejemplo, el archivo `std_msgs/msg/String.msg` tiene el tipo de mensaje `std_msgs/String`.
- **Repositorios:** Son colecciones de paquetes que comparten un sistema de control de versiones, por ejemplo, Git, Subversion, Mercurial y Bazaar.

B.0.3. Nivel de grafo de computación (*Computacional Graph Level*)

Este nivel se refiere a la representación en forma de grafo de la red basada en (*per-to-per*) que se encarga de procesar toda la información. La Figura B.2 representa el concepto de grafo de ROS y cómo se encuentra constituido con nodos, *topics*, mensajes, *master*, servidor de parámetros, servicios y *bags*.

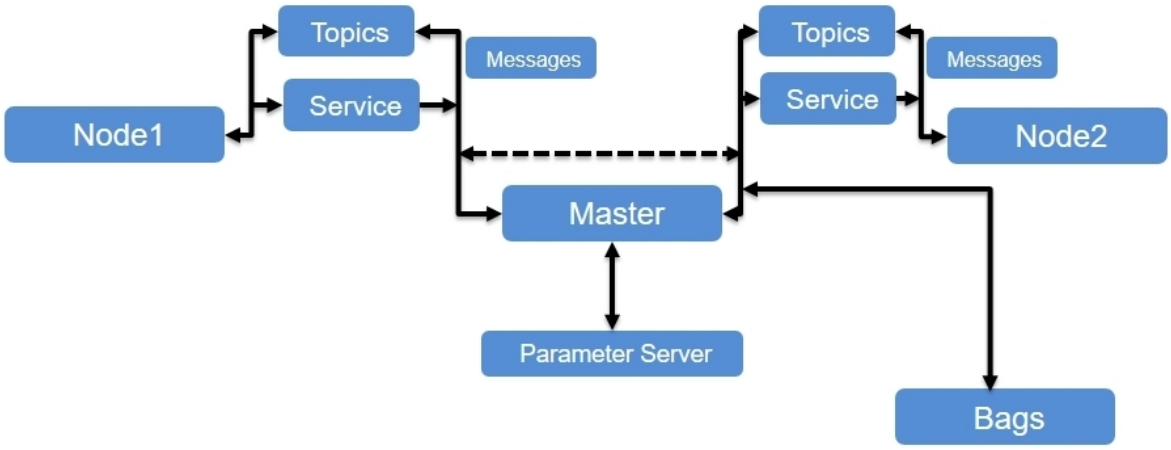


Figura B.2: Nivel de grafo de computación en ROS. Tomado de [131].

A continuación, se describe cada concepto correspondiente a este nivel:

- **Nodos:** Los nodos son procesos encargados de realizar los cálculos. Un nodo puede comunicarse con otros nodos a través de *topics*, servicios y parámetros. El uso de nodos en ROS proporciona ventajas, como la construcción de muchos procesos simples que cumplen funciones específicas en un sistema complejo. Existe tolerancia a fallas, ya que

los bloqueos se encuentran aislados en nodos individuales. Por la estructura simple de los nodos, se facilita su depuración. Para crear un nodo en ROS, se usa una biblioteca específica de cliente como *rospy* o *roscpp*. Con la herramienta de líneas de comando `rosnode`, es posible visualizar un listado de los nodos que se encuentran en ejecución.

- ***Master*:** Es el nodo principal. Proporciona registros de nombre y búsqueda para el resto de los nodos que componen el sistema. Sin un *master*, los nodos no podrán encontrarse, comunicarse, o solicitar servicios.
- **Servidor de parámetros:** Permite almacenar los datos en una localización central.
- **Mensajes:** Son una estructura de datos simple que contienen tipos de datos. Los nodos se comunican a través de la publicación de mensajes a *topics*.
- ***Topics*:** Los mensajes en ROS son transportados usando nombres de buses llamados *topics*. Cuando un nodo envía un mensaje a través de un *topic*, quiere decir que el nodo está publicando un *topic*. Cuando un nodo recibe un mensaje a través de un *topic*, significa que el nodo se está suscribiendo a un *topic*. Cada *topic* tiene un único nombre, y cualquier nodo puede acceder a este *topic* y enviar datos a través de él, siempre y cuando el tipo de mensaje sea el correcto.
- **Servicios:** Existen aplicaciones de robots, donde el modelo unidireccional de publicador/suscriptor no es suficiente, ya que se necesita una interacción solicitud/respuesta. Utilizando los servicios de ROS es posible dar solución a este tipo de casos, ya que se define un servicio que contenga dos partes; uno para solicitudes y el otro para respuestas. Para cada parte se puede escribir un nodo servidor y otro nodo cliente. En este caso el servidor proporciona el servicio bajo un nombre, y cuando el cliente envía el mensaje de solicitud al servidor, éste responderá y enviará el resultado al cliente.
- ***Bags*:** Se trata de un formato para guardar y reproducir datos de mensajes en ROS. Es un mecanismo importante que permite almacenar datos, de los sensores por ejemplo, que pueden ser difíciles de recopilar, pero que son necesarios para desarrollar y probar algoritmos.

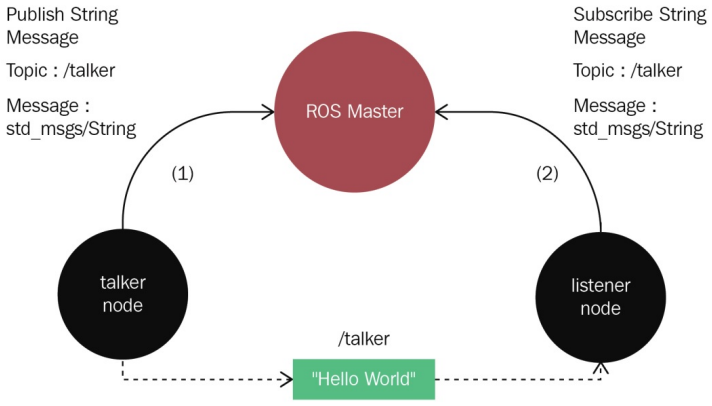


Figura B.3: Comunicación entre nodos en ROS. Tomado de [131].

La Figura B.3 representa la comunicación habitual entre dos nodos ROS, uno llamado *talker* y otro *listener*. El nodo *talker* se encarga de publicar un mensaje de tipo string `"Hello World"` en un *topic* denominado `/talker`, mientras que el nodo *listener* se suscribe a este *topic*.

B.0.4. Nivel de la comunidad

A este nivel pertenecen una serie de recursos que proporciona ROS, con los que habilita una comunidad que permite intercambiar aplicaciones de *software*, códigos y conocimiento.

- **Distribuciones**[59]**:** Al igual que ocurre en Linux, las distribuciones de ROS son una colección versionada de paquetes, han sido con el fin de proporcionar un conjunto base de *software* y código que sea estable para el desarrollo de aplicaciones de robots. Las distribuciones permiten instalaciones de ROS y paquetes de *software* de manera más sencilla y conjunta, como por ejemplo Gazebo.

Distribución ROS	**Fecha de lanzamiento**
Box Turtle	2 de marzo de 2010
C Turtle	2 de agosto de 2010
Diamondback	2 de marzo de 2011
Electric Emys	30 de agosto de 2011
Fuerte Trutle	23 de abril de 2012
Groovy Galapagos	31 de diciembre de 2012
Hydro Medusa	4 de septiembre de 2013
Indigo Igloo	22 de julio de 2014
Jade Turtle	23 de mayo de 2015
Kinetic Kame	23 de mayo de 2016
Lunar Loggerhead	23 de mayo de 2017
Melodic Morenia	23 de mayo de 2018
Noetic Ninjemys	23 de mayo de 2020

Tabla B.1: Distribuciones de ROS y fecha de lanzamiento.

- **Repositorios:** ROS depende de una red fedarada de códigos de repositorios, donde diferentes instituciones pueden desarrollar y lanzar sus propios componentes de *software* de robot.
- **Wiki ROS**[60]**:** Es el principal foro para documentar información sobre ROS. Cualquier persona puede registrarse, crear una cuenta y contribuir con su propia documentación, actualizaciones, correcciones, tutoriales, entre otros.
- **Sistema *ticket* de errores:** A través de este recurso es posible informar acerca de un error en el *software* o solicitar agregar una nueva característica al mismo.

[59] http://wiki.ros.org/Distributions
[60] http://wiki.ros.org/

- **Lista de correos:** Es el principal canal de comunicación con los usuarios ROS. Informa acerca de nuevas actualizaciones y cuenta con un foro para hacer preguntas.
- **Respuesta ROS:** Por medio de este recurso web se pueden publicar preguntas de ROS, visibles para otros usuarios que pueden dar respuesta y soluciones.
- **Blog**[61]**:** Este recurso contiene noticias, fotos y videos relacionados con la comunidad ROS.

Desde que ROS inició en el año 2007, la comunidad de robótica ha estado en constante cambio y contribuyendo con nuevas características para ROS. Estos cambios han permitido identificar limitaciones, que la comunidad de ROS ha tomado como justificación para el desarrollo de una nueva versión de ROS. Es así como ROS 2[62] se introdujo con una arquitectura renovada y con características mejoradas. ROS 2 aborda limitaciones en aspectos como la flexibilidad, comportamiento en tiempo real y seguridad en ROS, y abre el camino a una adopción más generalizada por parte de la industria y la academia. En la Tabla B.2 se muestra una lista de las distribuciones ROS 2 actuales e históricas[63].

Distribución ROS 2	**Fecha de lanzamiento**
Dashing Diademata	31 de mayo de 2019
Foxy Fitzroy	5 de junio de 2020
Galactic Geochelone	23 de mayo de 2021

Tabla B.2: Distribuciones de ROS 2 admitidas actualmente y fecha de lanzamiento.

[61] http://www.ros.org/news
[62] https://docs.ros.org/en/foxy/index.html
[63] https://docs.ros.org/en/foxy/Releases.html

C

Etiquetado del *dataset*

La tarea de anotación de imágenes consiste en dibujar un cuadro delimitador sobre el objeto que se planea detectar y asignar un nombre de etiqueta a la clase u objeto. La Figura 4.3 corresponde a una de las imágenes tomadas del *dataset* en LabelImg sobre la que se dibuja el cuadro delimitador y se le asigna la etiqueta previamente creada y nombrada en este caso como `'RC'`. Por cada imagen del *dataset* se realiza una anotación que se guarda como un *XML*, este archivo contiene las coordenadas del cuadro delimitador (*bounding box*) y el nombre de la etiqueta del objeto. Esta tarea de selección y anotación de imágenes es sencilla, sin embargo, es dispendiosa y requiere de una gran cantidad horas que dependen directamente del total de imágenes que conforman el *dataset*. Las anotaciones con el formato Pascal VOC, relaciona las coordenadas del *bounding box* como $(x_{min}, y_{min}, x_{max}, y_{max})$, la primera coordenada arriba a la izquierda y la segunda abajo a la derecha. Este formato es muy utilizado para la clasificación y detección de objetos. La clasificación permite determinar asignar y separar objetos de una clase en particular en una imagen, mientras que la detección y localización permite determinar dónde está el objeto en una imagen.

```
<annotation>
        <folder>Dataset</folder>
        <filename>Img1001.jpg</filename>
        <path>C:\Users\User\Pictures\Dataset\Img1001.jpg</path>
        <source>
                <database>Unknown</database>
        </source>
        <size>
                <width>540</width>
                <height>720</height>
                <depth>3</depth>
        </size>
        <segmented>0</segmented>
        <object>
                <name>RC</name>
```

```
16                  <pose>Unspecified</pose>
17                  <truncated>0</truncated>
18                  <difficult>0</difficult>
19                  <bndbox>
20                          <xmin>191</xmin>
21                          <ymin>310</ymin>
22                          <xmax>283</xmax>
23                          <ymax>463</ymax>
24                  </bndbox>
25          </object>
26  </annotation>
```

D

TensorRT

En 2018, NVIDIA anunció la integración de la herramienta de optimización de inferencias TensorRT con TensorFlow. A través de TensorRT, NVIDIA ofrece a los usuarios de TensorFlow, la posibilidad de aumentar el rendimiento de la inferencia para que sea lo más alto posible [132]. La Figura D.1 ilustra el flujo de trabajo para usar TensorRT, consiste en congelar el grafo de TensorFlow para obtener la inferencia. Luego, TensorRT realiza el trabajo de optimización de los subgrafos de TensorFlow, esto lo realiza reemplazando cada subgrafo compatible por un nodo optimizado de TensorRT. La salida, retorna un grafo congelado que se ejecuta para la inferencia de TensorFlow.

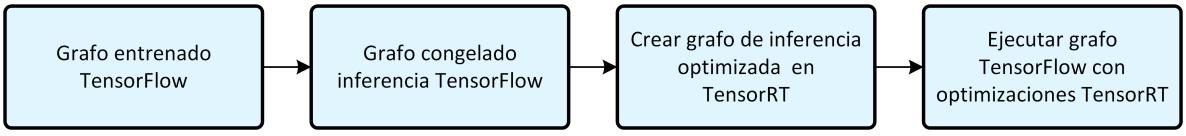


Figura D.1: Diagrama de flujo de integración de TensorRT en inferencia TensorFlow.

Las optimizaciones de TensorRT se aplican al grafo congelado de TensorFlow con la función de Python `create_inference_graph`. Este grafo es tomado como entrada por TensorRT y devuelve un grafo optimizado con nodos TensorRT.

```
1 trt_graph = trt.create_inference_graph(
2                 input_graph_def=frozen_graph_def,
3                 outputs=output_node_name,
4                 max_batch_size=batch_size,
5                 max_workspace_size_bytes=workspace_size,
6                 precision_mode=precision)
```

- `frozen_graph_def`: esta es la ruta de la carpeta para cargar el grafo de salida TensorFlow.

- `put_node_name`: esta es la ruta de la carpeta que contiene la lista de cadenas de caracteres con nombres de nodos de salida.
- `max_batch_size`: es un número de entero, su valor predeterminado es 1. Este es el tamaño máximo del lote de entrada.
- `max_workspace_size_bytes`: el valor por defecto es 1GB. Es el tamaño máximo de memoria GPU disponible para TensorRT durante su ejecución.
- `precision_mode`: cadena de caracteres, permite valores 'FP32', 'FP16' o 'INT8'.
- `minimum_segment_size`: este es el número mínimo de nodos necesarios para reemplazar un subgrafo.

Los parámetros de la función `create_inference_graph` para la optimización de las inferencias de TensorFlow, se configuraron de la siguiente forma:

```
trt_graph = trt.create_inference_graph(
                input_graph_def=frozen_graph_def,
                outputs=output_names,
                max_batch_size=1,
                max_workspace_size_bytes=1 << 26,
                precision_mode='FP16',
                minimum_segment_size=50)
```

Esta configuración, se llevó a cabo de acuerdo con la documentación oficial de NVIDIA y los ejemplos proporcionados en la guía de usuario para la aceleración de inferencia en TensorFlow con TensorRT (TF-TRT) [133]. Esta guía, brinda algunas buenas prácticas a tener en cuenta, una de ellas indica que la conversión del modelo TensorFlow al modelo optimizado debe realizarse sobre la máquina de destino, para este caso, la conversión se realizó directamente sobre el módulo Jetson TX2. Esto se debe a que TensorRT optimiza el grafo mediante el uso de las GPU disponibles, por lo tanto, es posible que el grafo optimizado no funcione correctamente en una GPU diferente.

A

Apéndice: Estructura del proyecto

```
/
├── models
└── model-1
    ├── eval
    ├── images
    │   ├── test
    │   │   ├── Img2918.jpg
    │   │   ├── Img2918.xml
    │   │   ├── ...
    │   │   ├── Img3403.jpg
    │   │   └── Img3403.xml
    │   ├── train
    │   │   ├── Img1001.jpg
    │   │   ├── Img1001.xml
    │   │   ├── ...
    │   │   ├── Img2917.jpg
    │   │   └── Img2917.xml
    │   ├── test_labels.csv
    │   └── train_labels.csv
    └── inference_graph
        ├── saved_model
        │   ├── variables
        │   └── saved_model.pb
        ├── checkpoint
        ├── frozen_inference_graph.pb
        ├── model.ckpt.data-00000-of-00001
        ├── model.ckpt.index
        └── model.ckpt.meta
```

```
├──
│   ├──
│   │   └── pipeline.config
│   ├── pre-trained-model
│   │   └── ssd_inception_v2_coco_2018_01_28
│   │       ├── saved_model
│   │       ├── checkpoint
│   │       ├── frozen_inference_graph.pb
│   │       ├── model.ckpt.data-00000-of-00001
│   │       ├── model.ckpt.index
│   │       ├── model.ckpt.meta
│   │       └── pipeline.config
│   ├── tensorboard
│   ├── training
│   ├── eval.py
│   ├── export_inference_graph.py
│   ├── generate_tfrecord.py
│   ├── test.record
│   ├── train.py
│   ├── train.record
│   └── xml_to_csv.py
├── model-2
├── model-3
├── results
├── velocity
├── video
├── labelmap.pbtxt
├── README.md
└── object_detection.py
```

Apéndice: Diseño y configuración de simulación

El entorno de simulación configurado en este trabajo es de tipo exterior y libre de obstáculos para el cuadricóptero. Para ejecutar una simulación Gazebo requiere cargar un entorno en él, por ello se necesita de un archivo que contenga la configuración del mundo. Este tipo de archivo tiene extensión `.world`, se encuentra escrito en XML y contiene los parámetros del motor físico *Open Dynamics Engine (ODE)*. Algunos de los parámetros ODE que se especifican en este archivo son: el paso de tiempo para cálculos dinámicos, el escalado interno para especificar cómo se manejan las dimensiones, la dinámica del mundo que se encarga de describir el estado de los cuerpos u objetos en un espacio 3D. En este archivo, también se definen parámetros utilizando el motor de gráficos orientados a objetos *Object Oriented Graphics Rendering Engine (OGRE)*, estos parámetros permiten que Gazebo proporcione una representación realista de entornos que incluyen, iluminación sombras y texturas de alta calidad.

El diseño y configuración del mundo que se usó para las simulaciones, se realizó con base en el archivo `outdoor_plane.world`. En lugar del plano montañoso que trae por defecto, se cambió por el modelo de un campo de cancha de fútbol robocup denominado `robocup_3dsim_field`, este modelo corresponde a uno de los mundos de `gazebo_worlds`. Por otro lado, el archivo *launch* `outdoor_gazebo.launch` es el encargado de realizar varias tareas, entre las que se encuentran: poner en marcha Gazebo y ejecutar el mundo previamente configurado, cargar el modelo y la simulación del robot cuadricóptero, adjuntar funciones de localización y trayectoria del robot, y por último, ejecución y visualización en RViz. A continuación se adjunta la configuración del archivo *launch* `outdoor_gazebo.launch` y las Figuras B.1, B.2, que ilustran el ambiente de simulación diseñado en RViz y Gazebo, respectivamente.

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <scene>
      <ambient>0.5 0.5 0.5 1</ambient>
      <background>0.5 0.5 0.5 1</background>
      <shadows>false</shadows>
    </scene>
    <physics type="ode">
      <gravity>0 0 -9.8</gravity>
      <ode>
        <solver>
          <type>quick</type>
          <iters>10</iters>
          <sor>1.3</sor>
        </solver>
        <constraints>
          <cfm>0.0</cfm>
          <erp>0.2</erp>
          <contact_max_correcting_vel>100.0</contact_max_correcting_vel>
          <contact_surface_layer>0.001</contact_surface_layer>
        </constraints>
      </ode>
      <real_time_update_rate>1000</real_time_update_rate>
      <max_step_size>0.001</max_step_size>
    </physics>
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://robocup_3Dsim_field</uri>
    </include>
    <light type="directional" name="my_light">
      <pose>0 0 30 0 0 0</pose>
      <diffuse>.5 .5 .5 1</diffuse>
      <specular>.1 .1 .1 1</specular>
      <attenuation>
        <range>200</range>
      </attenuation>
      <direction>0 0 -1</direction>
      <cast_shadows>false</cast_shadows>
    </light>
  </world>
</sdf>
```

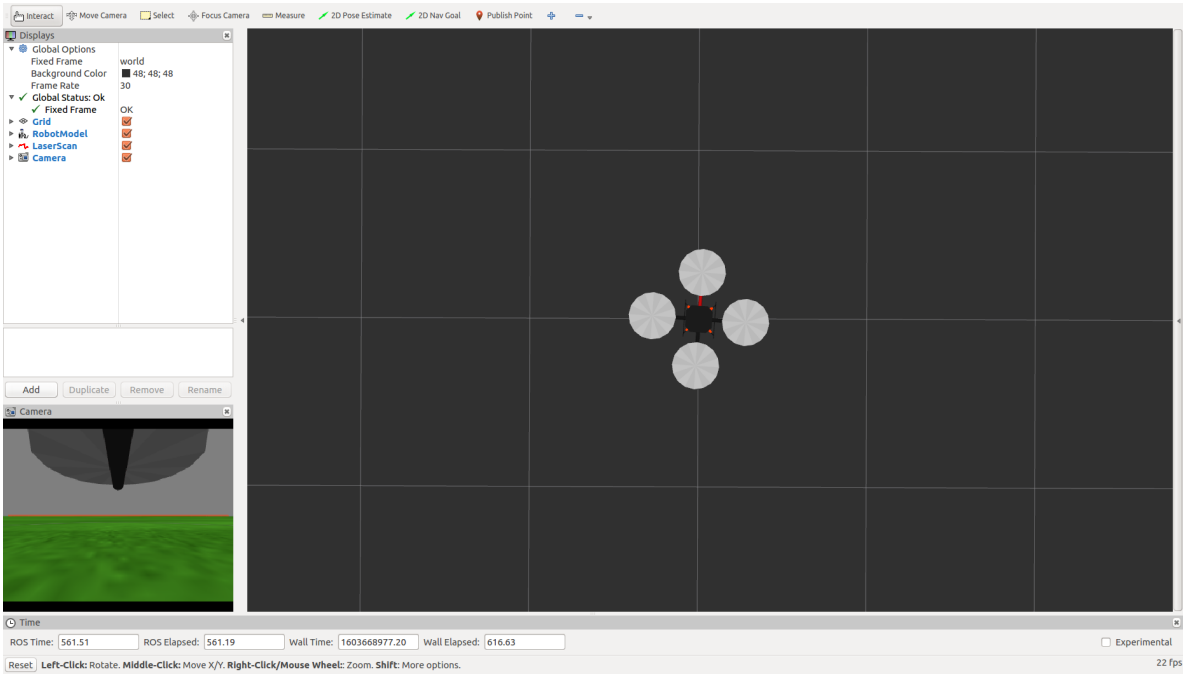


Figura B.1: Representación de ambiente exterior, libre de obstáculos en RViz.

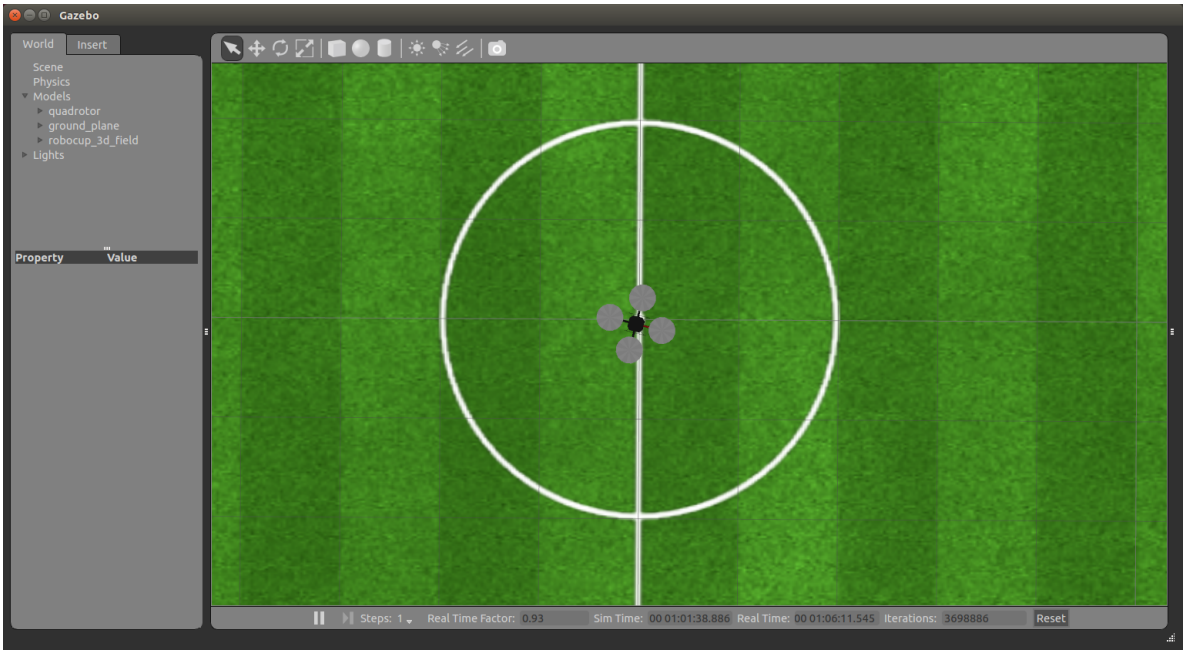


Figura B.2: Representación de ambiente exterior, libre de obstáculos en Gazebo.

Bibliografía

[1] Noah Shachtman. *Flying-Robot Cops, Farmers, and Oil Riggers Get to Work*. 2018. URL: https://www.wired.com/2012/06/ff-dronesatwork/.

[2] Luis F Luque-Vega y col. «Power line inspection via an unmanned aerial system based on the quadrotor helicopter». En: (2014), págs. 393-397.

[3] SP Yeong, LM King y SS Dol. «A review on marine search and rescue operations using unmanned aerial vehicles». En: *Int. J. Mech. Aerosp. Ind. Mech. Manuf. Eng* 9.2 (2015), págs. 396-399.

[4] Vladimir Sherstjuk, Maryna Zharikova e Igor Sokol. «Forest Fire Monitoring System Based on UAV Team, Remote Sensing, and Image Processing». En: (2018), págs. 590-594.

[5] Sonia Waharte y Niki Trigoni. «Supporting search and rescue operations with UAVs». En: (2010), págs. 142-147.

[6] Jonghyuk Kim. «Autonomous navigation for airborne applications». Tesis doct. Department of Aerospace, Mechanical y Mechatronic Engineering, 2004.

[7] Zhou Tao y Wang Lei. «SINS and GPS integrated navigation system of a small unmanned aerial vehicle». En: (2008), págs. 465-468.

[8] Amir Badshah y col. «Vehicle navigation in GPS denied environment for smart cities using vision sensors». En: *Computers, Environment and Urban Systems* 77 (2019), pág. 101281.

[9] Michael Blösch y col. «Vision based MAV navigation in unknown and unstructured environments». En: (2010), págs. 21-28.

[10] Tomáš Krajník y col. «A simple visual navigation system for an UAV». En: (2012), págs. 1-6.

[11] Yingcai Bi y Haibin Duan. «Implementation of autonomous visual tracking and landing for a low-cost quadrotor». En: *Optik-International Journal for Light and Electron Optics* 124.18 (2013), págs. 3296-3300.

[12] A. Shanthi y M.P. Sirisha. «Vehicle Detector Method for Complex Environments on Embedded Linux Platform». En: *International Journal of Computer Applications* (2013). URL: http://research.ijcaonline.org/volume78/number14/pxc3891363.pdf.

[13] Y. Watanabe, P. Fabiani y G. Le Besnerais. «Simultaneous visual target tracking and navigation in a GPS-denied environment». En: *Advanced Robotics, 2009. ICAR 2009. International Conference on*. Jun. de 2009, págs. 1-6.

[14] Jung-Ho Ahn y col. «Human tracking and silhouette extraction for human–robot interaction systems». En: *Pattern Analysis and Applications* 12.2 (2009), págs. 167-177. ISSN: 1433-7541. DOI: 10.1007/s10044-008-0112-3. URL: http://dx.doi.org/10.1007/s10044-008-0112-3.

[15] Michael H Kurniawan y col. «Human Anatomy Learning Systems Using Augmented Reality on Mobile Application». En: *Procedia Computer Science* 135 (2018). The 3rd International Conference on Computer Science and Computational Intelligence (ICCS-CI 2018) : Empowering Smart Technology in Digital Era for a Better Life, págs. 80 -88. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.08.152. URL: http://www.sciencedirect.com/science/article/pii/S1877050918314388.

[16] Madjid Maidi, Fakhreddine Ababsa y Malik Mallem. «Handling Occlusions for Robust Augmented Reality Systems». En: *EURASIP Journal on Image and Video Processing* 2010.1 (mayo de 2010), pág. 146123. ISSN: 1687-5281. DOI: 10.1155/2010/146123. URL: https://doi.org/10.1155/2010/146123.

[17] Dalle Patrice Jebali Maher y Mohamed Jemni. «Sign Language Recognition System Based on Prediction in Human-Computer Interaction». En: (2014), págs. 565-570.

[18] Zhijie Fang y Antonio M López. «Is the pedestrian going to cross? answering by 2d pose estimation». En: (2018), págs. 1271-1276.

[19] Singh Suriya Ghosh Anurag y CV Jawahar. «Towards Structured Analysis of Broadcast Badminton Videos». En: (2018), págs. 296-304.

[20] Ye Lyu y col. «The UAVid Dataset for Video Semantic Segmentation». En: *arXiv preprint arXiv:1810.10438* (2018).

[21] Chunhua Shen Zhongfei Zhang Anthony R. Dick Xi Li Weiming Hu y Anton van den Hengel. «A Survey of Appearance Models in Visual Object Tracking». En: *CoRR* abs/1303.4803 (2013). arXiv: 1303.4803. URL: http://arxiv.org/abs/1303.4803.

[22] J.E. Gomez-Balderas y col. «Tracking a Ground Moving Target with a Quadrotor Using Switching Control». English. En: *Journal of Intelligent and Robotic Systems* 70.1-4 (2013), págs. 65-78. ISSN: 0921-0296. DOI: 10.1007/s10846-012-9747-9. URL: http://dx.doi.org/10.1007/s10846-012-9747-9.

[23] Sara Gutiérrez Bermejo. «Modelado, simulación y control inteligente de UAVs». Valencia, España: Escuela Técnica Superior de Ingeniería, 2013.

[24] William Clayton Selby. «Autonomous navigation and tracking of dynamic surface targets on-board a computationally impoverished aerial vehicle». Tesis doct. Massachusetts Institute of Technology, 2011.

[25] Louisa Brooke-Holland. «Overview of military drones used by the UK armed forces». En: *House of Commons Library, Briefing Paper* 06493 (2015), pág. 11.

[26] Tom Bellemans Davy Janssens Muhammad Arsalan Khan Wim Ectors y Geert Wets. «UAV-Based Traffic Analysis: A Universal Guiding Framework Based on Literature Survey». En: *Transportation Research Procedia* 22 (2017). 19th EURO Working Group on Transportation Meeting, EWGT2016, 5-7 September 2016, Istanbul, Turkey, págs. 541 -550. ISSN: 2352-1465. DOI: https://doi.org/10.1016/j.trpro.2017.03.043. URL: http://www.sciencedirect.com/science/article/pii/S2352146517301783.

[27] Filipe Beretta y col. «Topographic modelling using uavs compared with traditional survey methods in mining». En: *REM-International Engineering Journal* 71.3 (2018), págs. 463-470.

[28] Sizov A. Tretyak O. Antropov V. Molitor N. Krzystek P. Briechle S. «UAV-based detection of unknown radioactive biomas deposits in Chernobyl's Exclusion Zone». En: *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences* 42.2 (2018).

[29] Hoppe Christof Daftry Shreyansh y Horst Bischof. «Building with drones: Accurate 3d facade reconstruction using mavs». En: (2015), págs. 3487-3494.

[30] Fiorenzo Franceschini, Luca Mastrogiacomo y Barbara Pralio. «An unmanned aerial vehicle-based system for large scale metrology applications». En: *International Journal of Production Research* 48.13 (2010), págs. 3867-3888.

[31] Eni Wardihani y col. «Real-time forest fire monitoring system using unmanned aerial vehicle». En: *Journal of Engineering Science and Technology* 13.6 (2018), págs. 1587-1594.

[32] Li Xiujuan Yu Junfeng Kumar-Mohit Zhang Weiping y Yihua Mao. «Remote sensing image mosaic technology based on SURF algorithm in agriculture». En: *EURASIP Journal on Image and Video Processing* 2018.1 (2018), pág. 85.

[33] Francisco Manuel Jiménez-Brenes y col. «Quantifying pruning impacts on olive tree architecture and annual canopy growth by using UAV-based 3D modelling». En: *Plant methods* 13.1 (2017), pág. 55.

[34] Castillo-Toledo Bernardino-Loukianov Alexander Luque-Vega Luis F. y Luis Enrique Gonzalez-Jimenez. «Power line inspection via an unmanned aerial system based on the quadrotor helicopter». En: *MELECON 2014-2014 17th IEEE Mediterranean Electrotechnical Conference*. IEEE. 2014, págs. 393-397.

[35] Uk Borey Konam-David Erdelj Milan y Enrico Natalizio. «From the Eye of the Storm: An IoT Ecosystem Made of Sensors, Smartphones and UAVs». En: *Sensors* 18.11 (2018), pág. 3814.

[36] Jasmina Pašagić Škrinjar, Pero Škorput y Martina Furdić. «Application of Unmanned Aerial Vehicles in Logistic Processes». En: *International Conference "New Technologies, Development and Applications"*. Springer. 2018, págs. 359-366.

[37] Nicola Roberto Zema, Enrico Natalizio y Evsen Yanmaz. «An unmanned aerial vehicle network for sport event filming with communication constraints». En: (2017).

[38] DroneDeploy. *DroneDeploy Selected by CNH Industrial for Intuitive New Drone System Targeting Ag Customers*. 2017. URL: https://medium.com/aerial-acuity/dronedeploy-selected-by-cnh-industrial-for-intuitive-new-drone-system-targeting-ag-customers-f275e83669e8.

[39] Samexa Corp. 2018. URL: https://www.prnewswire.com/news-releases/flying-robots-to-help-train-athletes-prevent-sports-injuries-248540121.html.

[40] Mary "Missy" Cummings. *Drones challenge legislation (Opinion)*. 2015. URL: https://edition.cnn.com/2015/01/29/opinion/cummings-drone-policies/index.html.

[41] *Dubai Launches World's First Rescue Drone, the "Flying Rescuer"*. 2018. URL: https://dubaiofw.com/dubai-flying-rescuer-drone/.

[42] Staff. *Cyberhawk Opens Asset Management Partnership for International UAV Powerline Inspection*. 2015. URL: https://dronelife.com/2015/09/08/cyberhawk-opens-asset-management-partnership-for-international-uav-powerline-inspection/.

[43] Christian Bettstetter. *Job selection in UAV-based delivery services*. 2014. URL: https://bettstetter.com/job-selection-uav/.

[44] Miniterio de Economía y Competitividad de España. «Vehículos aéreos no tripulados (UAVs)». En: Madrid, España, 2013.

[45] Alvika Gautam, PB Sujit y Srikanth Saripalli. «A survey of autonomous landing techniques for UAVs». En: (2014), págs. 1210-1218.

[46] Yi Feng y col. «Autonomous landing of a UAV on a moving platform using model predictive control». En: *Drones* 2.4 (2018), pág. 34.

[47] Patrick Benavidez y col. «Landing of an Ardrone 2.0 quadcopter on a mobile base using fuzzy logic». En: (2014), págs. 803-812.

[48] David Cabecinhas y col. «Robust take-off and landing for a quadrotor vehicle». En: (2010), págs. 1630-1635.

[49] Youeyun Jung, Hyochoong Bang y Dongjin Lee. «Robust marker tracking algorithm for precise UAV vision-based autonomous landing». En: (2015), págs. 443-446.

[50] Mengyin Fu y col. «Autonomous landing of a quadrotor on an UGV». En: (2016), págs. 988-993.

[51] Xuqiang Zhao, Qing Fei y Qingbo Geng. «Vision based ground target tracking for rotor UAV». En: (2013), págs. 1907-1911.

[52] Iván Fernando Mondragón y col. «Visual model feature tracking for UAV control». En: (2007), págs. 1-6.

[53] Xiyan Chen y Qinggang Meng. «Vehicle detection from UAVs by using SIFT with implicit shape model». En: (2013), págs. 3139-3144.

[54] Xi Chao-jian y Guo San-xue. «Image target identification of UAV based on SIFT». En: *Procedia Engineering* 15 (2011), págs. 3205-3209.

[55] R Om Prakash y Chandran Saravanan. «Autonomous robust helipad detection algorithm using computer vision». En: (2016), págs. 2599-2604.

[56] Cheng Cheng, Xuzhi Wang y Xiangjie Li. «UAV image matching based on surf feature and harris corner algorithm». En: (2017).

[57] Songyun Xie y col. «Fast detecting moving objects in moving background using ORB feature matching». En: (2013), págs. 304-309.

[58] Pablo Ramon Soria, Begoña C Arrue y Anibal Ollero. «Detection, location and grasping objects using a stereo sensor on uav in outdoor environments». En: *Sensors* 17.1 (2017), pág. 103.

[59] Patrick Karlsson y Emil Johansson. «Object Identifier System for Autonomous UAV: A subsystem providing methods for detecting and descending to an object. The object is located in a specified area with a coverage algorithm.» En: (2018).

[60] Tian Xiang y col. «UAV based target tracking and recognition». En: (2016), págs. 400-405.

[61] Nassim Ammour y col. «Deep learning approach for car detection in UAV imagery». En: *Remote Sensing* 9.4 (2017), pág. 312.

[62] Giuseppe Amato y col. «Counting vehicles with deep learning in onboard uav imagery». En: (2019), págs. 1-6.

[63] Shubo Wang y col. «A Deep-Learning-Based Sea Search and Rescue Algorithm by UAV Remote Sensing». En: (2018), págs. 1-5.

[64] Srishti Srivastava, Sarthak Narayan y Sparsh Mittal. «A survey of deep learning techniques for vehicle detection from UAV images». En: *Journal of Systems Architecture* (2021), pág. 102152.

[65] Cong Wang y col. «Research of UAV target detection and flight control based on deep learning». En: (2018), págs. 170-174.

[66] Wei Zhang y col. «Coarse-to-fine uav target tracking with deep reinforcement learning». En: *IEEE Transactions on Automation Science and Engineering* 16.4 (2018), págs. 1522-1530.

[67] Jose-Ernesto Gomez-Balderas y col. «Tracking a ground moving target with a quadrotor using switching control». En: *Journal of Intelligent & Robotic Systems* 70.1-4 (2013), págs. 65-78.

[68] Kamel Boudjit y C Larbes. «Detection and target tracking with a quadrotor using fuzzy logic». En: (2016), págs. 127-132.

[69] Syaril Azrad, Farid Kendoul y Kenzo Nonami. «Visual servoing of quadrotor micro-air vehicle using color-based tracking algorithm». En: *Journal of System Design and Dynamics* 4.2 (2010), págs. 255-268.

[70] Celine Teuliere, Laurent Eck y Eric Marchand. «Chasing a moving target from a flying UAV». En: (2011), págs. 4929-4934.

[71] TK Venugopalan, Tawfiq Taher y George Barbastathis. «Autonomous landing of an Unmanned Aerial Vehicle on an autonomous marine vehicle». En: (2012), págs. 1-9.

[72] Pedro Vílez, Novel Certad y Elvis Ruiz. «Trajectory generation and tracking using the AR. Drone 2.0 quadcopter UAV». En: (2015), págs. 73-78.

[73] Salman Khan y col. «A guide to convolutional neural networks for computer vision». En: *Synthesis Lectures on Computer Vision* 8.1 (2018), págs. 1-207.

[74] M. Elgendy. *Deep Learning for Vision Systems*. Manning Publications, 2020. ISBN: 9781617296192. URL: https://books.google.com.co/books?id=6gkLzAEACAAJ.

[75] Benjamin Planche. *Hands-On Computer Vision with TensorFlow 2*. Packt Publishing, 2019.

[76] Adrian Rosebrock. *Deep Learning for Computer Vision with Python: ImageNet Bundle*. PyImageSearch, 2017.

[77] Oliver Theobald. *Machine learning for absolute beginners*. 2017.

[78] Jinsu Lee, Sang-Kwang Lee y Seong-Il Yang. «An ensemble method of cnn models for object detection». En: (2018), págs. 898-901.

[79] Yiming Zhang, Xiangyun Xiao y Xubo Yang. «Real-Time Object Detection for 360-Degree Panoramic Image Using CNN». En: (2017), págs. 18-23.

[80] BN Krishna Sai y T Sasikala. «Object Detection and Count of Objects in Image using Tensor Flow Object Detection API». En: (2019), págs. 542-546.

[81] Kingsley Kuan y col. «Region average pooling for context-aware object detection». En: (2017), págs. 1347-1351.

[82] Alif Bin Abdul Qayyum, Asiful Arefeen y Celia Shahnaz. «Convolutional Neural Network (CNN) Based Speech-Emotion Recognition». En: (2019), págs. 122-125.

[83] Hoseong Lee. *Deep learning object detection*. URL: https://github.com/hoya012/deep_learning_object_detection (visitado 04-08-2020).

[84] Ross Girshick y col. «Rich feature hierarchies for accurate object detection and semantic segmentation». En: (2014), págs. 580-587.

[85] Ross Girshick. «Fast R-CNN». En: (2015), págs. 1440-1448.

[86] Shaoqing Ren y col. «Faster R-CNN: Towards real-time object detection with region proposal networks». En: (2015), págs. 91-99.

[87] Wei Liu y col. «SSD: Single Shot Multibox Detector». En: (2016), págs. 21-37.

[88] Karen Simonyan y Andrew Zisserman. «Very deep convolutional networks for large-scale image recognition». En: *arXiv preprint arXiv:1409.1556* (2014).

[89] Zhang Xuexi y col. «SLAM algorithm analysis of mobile robot based on lidar». En: (2019), págs. 4739-4745.

[90] Hesham Ibrahim Mohamed Ahmed Omara y Khairul Salleh Mohamed Sahari. «Indoor mapping using kinect and ROS». En: (2015), págs. 110-116.

[91] Ilmir Z Ibragimov e Ilya M Afanasyev. «Comparison of ROS-based visual SLAM methods in homogeneous indoor environment». En: (2017), págs. 1-6.

[92] Carol Fairchild y Thomas L. Harman. *ROS Robotics By Example: Learning to control wheeled, limbed, and flying robots using ROS Kinetic Kame*. Packt Publishing Ltd, 2017.

[93] Kenta Takaya y col. «Simulation environment for mobile robots testing using ROS and Gazebo». En: (2016), págs. 96-101.

[94] Pratik Vyavahare, Sivaranjani Jayaprakash y Krishna Bharatia. «Construction of URDF model based on open source robot dog using Gazebo and ROS». En: (2019), págs. 1-5.

[95] Rajesh Kannan Megalingam y col. «ROS Based, Simulation and Control of a Wheeled Robot using Gamer's Steering Wheel». En: (2018), págs. 1-5.

[96] Gary Bradski y Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. Cambridge, MA: O'Reilly, 2008.

[97] Rasika Phadnis, Jaya Mishra y Shruti Bendale. «Objects talk-object detection and pattern tracking using TensorFlow». En: (2018), págs. 1216-1219.

[98] Jonathan Huang y col. «Speed accuracy trade-offs for modern convolutional object detectors». En: (2017), págs. 7310-7311.

[99] Tsung-Yi Lin y col. «Microsoft COCO: Common objects in context». En: *European conference on computer vision*. Springer. 2014, págs. 740-755.

[100] Andreas Geiger y col. «Vision meets robotics: The KITTI dataset». En: *The International Journal of Robotics Research* 32.11 (2013), págs. 1231-1237.

[101] Alina Kuznetsova y col. «The open images dataset v4». En: *International Journal of Computer Vision* (2020), págs. 1-26.

[102] Chunhui Gu y col. «AVA: A video dataset of spatio-temporally localized atomic visual actions». En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, págs. 6047-6056.

[103] NVIDIA Corporation. *NVIDIA Deep Learning Plataform: Giant Leaps in Performance and Efficiency for AI Services, From the Data Center to the Network's Edge*. Dic. de 2017. URL: http://www.nextplatform.com/wp-content/uploads/2018/01/inference-technical-overview-1.pdf (visitado 27-06-2020).

[104] Rengan Xu, Frank Han y Quy Ta. «Deep learning at scale on NVIDIA V100 accelerators». En: (2018), págs. 23-32.

[105] NVIDIA Corporation. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone (visitado 27-06-2020).

[106] Bhaumik Vaidya. *Hands-On GPU-Accelerated Computer Vision with OpenCV and CUDA: Effective techniques for processing complex image data in real time using GPUs*. Packt Publishing Ltd, 2018.

[107] NVIDIA Corporation. *NVIDIA cuDNN*. URL: https://developer.nvidia.com/cudnn (visitado 27-06-2020).

[108] Kaiming He y col. «Deep residual learning for image recognition». En: (2016), págs. 770-778.

[109] Saining Xie y col. «Aggregated residual transformations for deep neural networks». En: (2017), págs. 1492-1500.

[110] Kaiming He y col. «Mask R-CNN». En: (2017), págs. 2961-2969.

[111] Olaf Ronneberger, Philipp Fischer y Thomas Brox. «U-net: Convolutional networks for biomedical image segmentation». En: (2015), págs. 234-241.

[112] Fausto Milletari, Nassir Navab y Seyed-Ahmad Ahmadi. «V-net: Fully convolutional neural networks for volumetric medical image segmentation». En: (2016), págs. 565-571.

[113] Jacob Devlin y col. «Bert: Pre-training of deep bidirectional transformers for language understanding». En: *arXiv preprint arXiv:1810.04805* (2018).

[114] Alec Radford y col. «Language models are unsupervised multitask learners». En: *OpenAI Blog* 1.8 (2019), pág. 9.

[115] Ryan Prenger, Rafael Valle y Bryan Catanzaro. «Waveglow: A flow-based generative network for speech synthesis». En: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2019, págs. 3617-3621.

[116] *cuDNN Support Matrix*. Abr. de 2021. URL: https://docs.nvidia.com/deeplearning/cudnn/support-matrix/index.html.

[117] Vivienne Sze y col. «Hardware for machine learning: Challenges and opportunities». En: (2017), págs. 1-8.

[118] Maurizio Capra y col. «An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks». En: *Future Internet* 12.7 (2020), pág. 113.

[119] Kenneth C Laudon y Jane Price Laudon. *Sistemas de información gerencial: administración de la empresa digital*. Pearson Educación, 2004, pág. 507.

[120] Sendobry Alexander Kohlbrecher Stefan Klingauf-Uwe Meyer Johannes y Oskar von Stryk. «Comprehensive Simulation of Quadrotor UAVs Using ROS and Gazebo». En: (2012), págs. 400-411.

[121] Xiaofei Wang y col. «Convergence of edge computing and deep learning: A comprehensive survey». En: *IEEE Communications Surveys & Tutorials* 22.2 (2020), págs. 869-904.

[122] *NVIDIA TensorRT Documentation*. Feb. de 2021. URL: https://docs.nvidia.com/deeplearning/tensorrt/support-matrix/index.html#hardware-precision-matrix.

[123] NVIDIA Corporation. *NVIDIA Jetson Linux Driver Package Software Features. Power Management for Jetson TX2 Series Devices*. URL: https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%2520Linux%2520Driver%2520Package%2520Development%2520Guide%2Fpower_management_tx2_32.html%23wwpID0EXHA (visitado 01-10-2020).

[124] Lukas Cavigelli. *Convenient Power Measurements on the Jetson TX2/Tegra X2 Board*. URL: https://embeddedml.wordpress.com/2018/04/25/convenient-power-measurements-on-the-jetson-tx2-tegra-x2-board/ (visitado 26-11-2020).

[125] Diane K Fisher y Alexander Novati. «Make a pinhole camera». En: *Technology and Engineering Teacher* 69.3 (2009), pág. 15.

[126] Aaron Martinez y Enrique Fernndez. *Learning ROS for Robotics Programming*. Packt Publishing, 2013. ISBN: 1782161449, 9781782161448.

[127] Morgan Quigley y col. «ROS: an open-source Robot Operating System». En: *ICRA Workshop on Open Source Software*. 2009.

[128] *Robot Operating System*. URL: http://www.ros.org/ (visitado 01-06-2019).

[129] Bernardo Ronquillo Japón. *Hands-On ROS for Robotics Programming: Program highly autonomous and AI-capable mobile robots powered by ROS*. Packt Publishing, 2020.

[130] Lentin Joseph y Jonathan Cacace. *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System*. Packt Publishing Ltd, 2018.

[131] Lentin Joseph. *ROS Robotics Projects*. Packt Publishing Ltd, 2017.

[132] *TensorRT Integration Speeds Up TensorFlow Inference*. Ago. de 2020. URL: https://developer.nvidia.com/blog/tensorrt-integration-speeds-tensorflow-inference/.

[133] *NVIDIA TensorRT Documentation*. Feb. de 2021. URL: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.